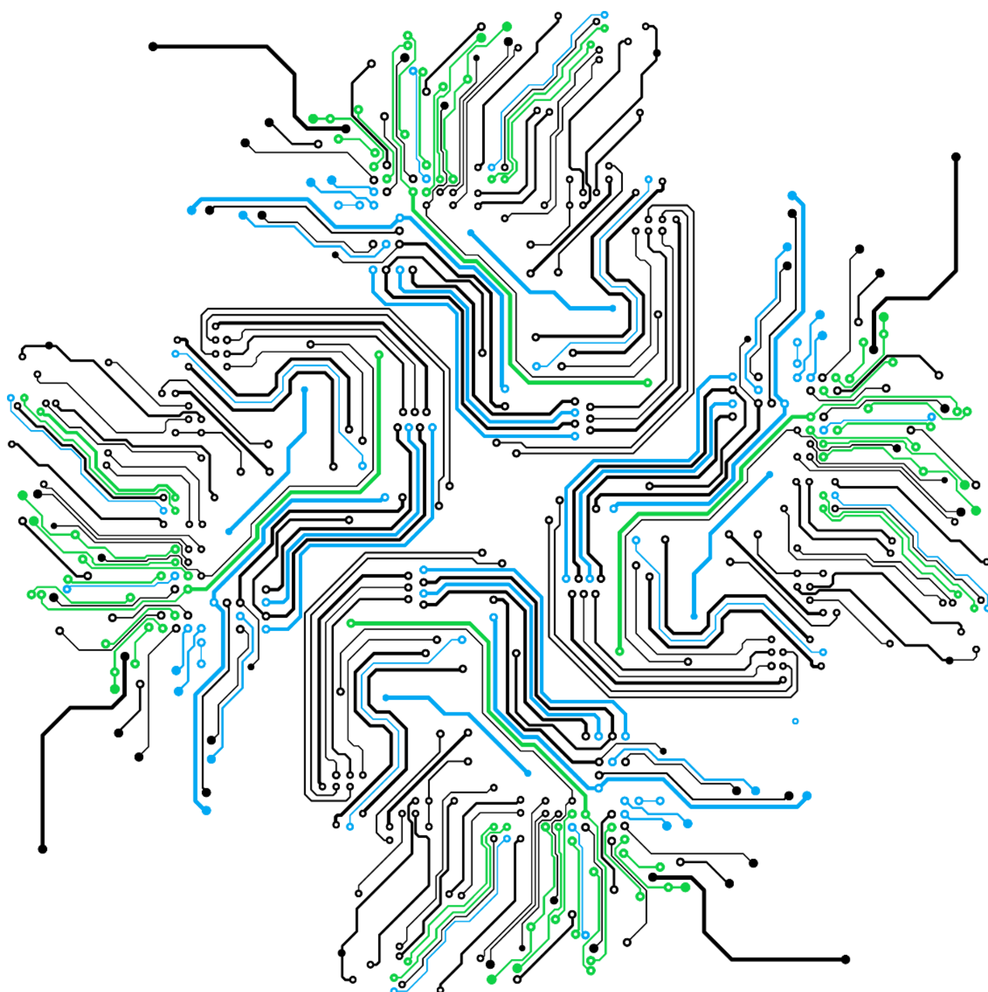
Juho Keränen

Ohjelmistokehitysympäristön suunnittelu ja toteutus



Insinööri (AMK)

Tieto- ja viestintätekniikka

Kevät 2019

KAMK • University of Applied Sciences

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Keränen Juho

Työn nimi: Ohjelmistokehitysympäristön suunnittelu ja toteutus

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: analysaattori, IO-alusta, kehitysympäristö, ohjelmistonkehitys

Työn tilaajana oli Valmet Oyj:n Kajaanin yksikkö Valmet Automation Oy. Valmet on suomalainen pörssiyhtiö, joka toimittaa teknologiaa, automaatiota ja palveluita sellu-, paperi- ja energiateollisuuden alalle. Valmetin automaatio-liiketoimintalinja tarjoaa asiakkaille automaatioratkaisuja. Kajaanin yksikön päätuotteina ovat analysaattorit ja mittalaitteet.

Työn tavoitteena oli tutkia, suunnitella ja kehittää Valmetin analysaattoreissa käytettävän uuden IO-alustan kehitysympäristö. IO-alustan tarkoitus on ohjata laitteiden lähtöjä ja tuloja ja hoitaa laitteen tietoliikenne.

Raportin alussa käsitellään ohjelmistoarkkitehtuurin ja ohjelmistontuotannon teoriaa. Ohjelmistojen kehittäminen on monivaiheinen prosessi, etenkin sulautettujen laitteiden kanssa. Sulautettuja laitteita suunniteltaessa ja niille ohjelmistoja kehittäessä tulee huomioida useita eri asioita, jotka vaikuttavat paljon lopputulokseen ja laitteen tulevaisuuteen.

Työssä hyödynnettiin kehitysalustaa, jonka avulla voidaan helposti testata haluttuja ominaisuuksia ennen oman kortin suunnittelua. Toteutusvaiheessa käydään koko prosessi läpi: vaatimusmäärittely, suunnittelu, toteutus ja testaus.

Kehitysympäristöä testattiin käyttämällä mikrokontrollerin sisältävää kehitysalustaa ohjelmiston ajamiseen. Tulosten perusteella havaittiin, että kehitysympäristö toimii asianmukaisesti. Kehitysympäristön käytön helpottamiseksi erillisten kirjastojen suunnitteleminen toteutetaan myöhemmin jatkokehityksessä.

Lopputuloksena on kehitysympäristö, jossa on tarvittavat komponentit ohjelmistojenkehitystyötä varten ja ohjeistus siitä, kuinka kehitysympäristöä käytetään.

KAMK • University of Applied Sciences

Abstract

Author(s): Keränen Juho

Title of the Publication: Software Development Environment Design and Implementation

Degree Title: Bachelor of Engineering, Information and communication technologies

Keywords: analyzer, IO platform, development environment, software development

This thesis was commissioned by Valmet Oyj, a unit of Valmet Automation Oy located in Kajaani, Finland. Valmet is a Finnish corporation that delivers technology, automation and services to the pulp, paper and energy industries. Valmet Automation business line offers automation solutions to customers. The main products of the Kajaani unit are analyzers and measuring instruments.

The purpose of this thesis was to research, develop and design the development environment for the new IO platform used in Valmet analyzers. The purpose of the IO platform is to control the inputs and outputs of the devices and to manage the data communications of the device.

The thesis details theory behind software architecture and software development. Software development is a multi-stage process, especially with embedded devices. When designing embedded devices and developing software for them, several things must be considered, that will greatly affect the outcome and future of the device.

In this thesis, a development platform was utilized to easily test the desired features before designing the prototype board. The development environment design process was thoroughly reviewed in the implementation part of this thesis: requirements, planning, development and testing.

The development environment was tested in the final development phase, using the microcontroller development platform to run the software. The results show that the development environment works as it should. To facilitate the use of the development environment, the design of separate libraries will be implemented later in further development.

The result of this thesis is an operational development environment, which has all the components needed for developing software, and instruction on how to use the development environment.

Sisällys

1 Johdanto ... 1

2 Ohjelmistoarkkitehtuuri ... 3

2.1 Ohjelmistoarkkitehtuurin määritelmä ... 3

2.2 Arkkitehtuuri ohjelmistonkehityksessä ... 4

3 Ohjelmistotuotanto ... 6

3.1 Ohjelmiston elinkaari ... 6

3.2 Ohjelmiston ominaisuuksia ... 9

3.3 Ohjelmistonkehitys projektina ... 10

3.3.1 Asiakkaan kannalta ... 10

3.3.2 Toimittajan kannalta ... 12

3.3.3 Projektityön kulku ... 13

3.4 Ohjelmiston uudelleenkäyttö ... 14

4 Sulautettu järjestelmä ... 16

4.1 Sulautetun järjestelmän suunnittelu ... 16

4.2 Sulautettu elektroniikka ... 18

4.2.1 Mikroprosessori ... 18

4.2.2 Mikrokontrolleri ... 19

4.2.3 Järjestelmäpiiri ... 21

4.3 Sulautettu ohjelmointi ... 22

4.3.1 Kehitysympäristö ... 24

4.3.2 Virheet ... 24

4.3.3 Testaus ... 25

4.3.4 Ohjelmiston laitteistorajapinta ... 26

4.3.5 Ohjelmiston siirrettävyys ... 28

5 IO-kortin esittely ... 30

5.1 ARM-mikrokontrollerin ominaisuuksia ... 31

5.2 Mikrokontrolleri ... 32

5.2.1 Mikrokontrollerin rajapinta ... 35

5.2.2 ARM-rajapinta ... 36

5.3 Tietoliikenne ... 37

6 Kehitysympäristön toteutus 41
6.1 Vaatimusmäärittely 41
6.2 Suunnittelu 41
6.3 Kehitys 43
6.3.1 Ohjelmointiympäristö 44
6.3.2 Ohjelmiston ja laitteiston rajapinta 47
6.3.3 Versionhallinta 48
6.3.4 Tietoliikenne 49
6.3.5 PC-simulaatio 49
6.4 Tietoliikennekirjaston muokkaus 50
6.5 Testaus 52
6.5.1 Käytettävyystestaus 53
6.5.2 Integraatiotestaus 53

7 Tulokset 54
7.1 Tulosten analysointi 54
7.1.1 Käytettävyys 54
7.1.2 Integraatio 59
7.2 Pohdinta 59
7.3 Jatkokehitys 60

8 Yhteenveto 61

Lähteet 62

Symboliluettelo

A/D	Analog-to-Digital, analogisen signaalin muuntaminen digitaaliseksi.
AMD	Advanced Micro Devices, Yhdysvaltalainen puolijohdevalmistaja.
ANSI	American National Standards Institute, Yhdysvaltalainen organisaatio joka valvoo standardien kehittymistä Yhdysvalloissa.
API	Application programming interface, ohjelmointirajapinta, jonka avulla ohjelmat keskustelevat keskenään.
ARM	Advanced RISC Machines, 32-bittinen mikroprosessoriarkkitehtuuri.
CMSIS	Cortex Microcontroller Software Interface Standard, valmistaja riippumaton laitteistorajapinta.
GPIO	General Purpose I/O, mikrokontrollerissa olevia pinnejä jotka voi ohjelmoida lähettämään tai vastaanottamaan signaaleja.
HAL	Hardware Abstraction Layer, rajapinta joka erottaa laitteiston ja ohjelmiston toisistaan.
IDE	Integrated Development Environment, ohjelmointiympäristö.
IEEE	Institute of Electrical and Electronics Engineers, kansainvälinen tekniikan alan järjestö.
I/O	Input/Output, tiedon siirtämistä laitteistojen välillä.
JTAG	Joint Test Action Group, ohjelmiston ja laitteiston testaukseen käytettävä liitäntä.

MAC	Media Access Control, verkkosovittimen yksilöivä osoite.
Modbus	Modiconin julkaisema sarjaliikenneprotokolla.
OSAL	Operating System Abstraction Layer, käyttöjärjestelmä rajapinta.
OSI-malli	Open Systems Interconnection Reference Model, tiedonsiirtoprotokollien kuvaamiseen käytettävä tekniikka.
RISC	Reduced Instruction Set Computer, suoritinarkkitehtuurien suunnittelufilosofia.
SPI	Serial Peripheral Interface, synkroninen sarjamuotoinen oheislaiteväylä.
SVN	Subversion, versionhallintajärjestelmä.
TCP	Transmission Control Protocol, tietoliikenneprotokolla.
TCP/IP	Transmission Control Protocol/Internet Protocol, usean Internet-liikennöinnissä käytettävän tietoliikenneprotokollan yhdistelmä.
UART	Universal Asynchronous Receiver Transmitter, asynkroninen sarjamuotoinen oheislaiteväylä.
UDP	User Datagram Protocol, yhteydetön tiedonsiirtoprotokolla

1 Johdanto

Ohjelmiston kehitystyö on jokapäiväinen haaste nykypäivänä kaikkialla, varsinkin teollisuudessa. Teollisuudessa etsitään taukoamatta uusia menetelmiä tuotannon kehittämiseksi ja pyritään kustannustehokkaisiin, luotettaviin ja kilpailukykyisiin tuotteisiin. Ohjelmistonkehityksellä on tässä suuri merkitys, koska ohjelmiston osuus voi olla jopa yli 50 % laitteen kehitys- ja ylläpitokustannuksista.

Työn tilaajana toimii Valmet Oyj:n Kajaanin yksikkö Valmet Automation Oy. Valmet on suomalainen pörssiyhtiö, joka toimittaa teknologia- ja automaatioratkaisuja ja huoltopalveluita sellu-, paperi- ja energiateollisuuden alalle. Valmetilla on yli 200 vuoden teollisuushistoria, ja työntekijöitä yli 12 000. Valmetin pääkonttori sijaitsee Espoossa, mutta maantieteellisesti Valmetin toiminnot jakaantuvat ympäri maailman, 33 maahan, joissa yhteensä 130 toimipistettä. [1.]

Valmet Automation Oy:n Kajaanin organisaatio jakaantuu useampaan eri osastoon. Organisaatioon kuuluu tuotekehitys, myynti ja markkinointi, huolto ja tuotetuki ja tuotanto. Kajaanin tuotekehitysyksikössä kehitetään analysaattoreita, sensoreita ja lähettimiä, joita hyödynnetään teollisuudessa. Näillä laitteilla pyritään parantamaan tuotannon kustannus-, energia- ja materiaalitehokkuutta. Laitteen yhteydessä Kajaanin yksikkö yleensä toimittaa valmiin integraation tehtaan prosessiin, eli automaattisen näytteenoton ja laite myös liitetään osaksi tehtaan tietojärjestelmää. Ylintä abstraktiotasoa toimituksissa edustavat valmiit säätösovellutukset, esimerkiksi keiton- ja valkaisun säätö kemiallisessa massan valmistuksessa tai jauhatuksen säätö TMP-laitoksella.

Mittalaitteilla mitataan halutuista kohdista tehtaan prosessia suure tai joukko suureita. Mittauksen tarkoituksena on yleensä prosessin ohjaus ja stabilointi, joissain tapauksissa myös tuotesertifiointi tai prosessin parantaminen ja tutkimuskäyttö. Mittauksen kohteena oleva asia tai ilmiö voi vaatia halutun arvon laskemiseen hyvinkin vaikeita algoritmeja, mutta yksinkertaisimmillaan se voi olla vaikka esineen pituuden mittaamista rullamitalla. [2.]

Analysaattoreilla suoritetaan analyysejä näytteistä. Näytteet voivat koostua jonkinlaisesta aineesta, kuten nesteestä, kiinteästä aineesta tai kaasusta. Analysaattorit koostuvat useasta eri osasta ja moduulista, jolloin jokaisella moduulilla on oma tehtävänsä laitteen kokoonpanossa. Tällaista kokoonpanoa ohjataan elektroniikalla, joka tyypillisesti koostuu Linux tietokoneesta ja

joukosta IO-kortteja. IO-kortit ohjaavat edelleen laitteen toimilaitteita ja niihin kytketään myös mittausanturien signaalit.

Perusperiaatteeltaan IO-kortin on tarkoitus ohjata tuloja ja lähtöjä niin, että tuloporttien kautta analysaattorin logiikka saa tietoa jonkin järjestelmän tilasta ja taas lähtöporttien kautta se voi ohjata järjestelmiä. IO-kortti sisältää myös tietoliikennerajapinnan, jonka avulla se keskustelee laitetta ohjaavan Linux-tietokoneen kanssa. Tämä kommunikaatiorajapinta edustaa tyypillisesti IO:n senhetkistä tilaa.

Tekniikan kehittyessä ja teollisuuden hakiessa parannuksia koko ajan myös analysaattoreiden täytyy kehittyä moduuleineen. Uusien analysaattorisukupolvien kehitystyössä on tarkoitus hyödyntää uutta mikrokontrolleriperhettä, jolloin IO-korttien suunnitteluympäristö ja työkalut muuttuvat. Tämän työn tavoitteena on tutkia ja parantaa analysaattoreiden seuraavan IO-korttisukupolven ohjelmistonkehitystä, eli luoda pohja uusien IO-korttien ja moduulien ohjelmistokehitykseen, niin että samaa ohjelmistoa voidaan hyödyntää kaikissa IO-moduuleissa ja muissa mittalaitteissa. Tässä raportissa keskitytään uuden mikrokontrollerin toimintaan ja siihen, kuinka sitä hyödynnetään, ohjelmistonkehitystyöhön ja siihen, minkälaisia komponentteja ohjelmistonkehitysympäristössä tarvitaan, etenkin sulautetun raudan ohjelmoinnissa.

Työ toteutetaan suunnittelemalla uusi kehitysympäristö, joka sisältää kehitystyöhön tarvittavat ohjelmistot, koodit ja dokumentit. Tämä sisältää eri vaihtoehtojen testauksen, evaluoinnin ja perustelun. Työhön sisältyy myös elektroniikan perusvalinnat yhdessä sen alan asiantuntijoiden kanssa, niin että ne mahdollistavat helpon IO-ohjelmistokehityksen. Lopputuloksena saadaan toimiva, testattu ja valmis kehitysympäristö, jolla ohjelmistonkehittäjät voivat helposti aloittaa uuden mikrokontrolleriperheen ohjelmistokehitystyön.

Seuraavassa osassa tutustutaan arkkitehtuurin merkitykseen ohjelmistoissa, jonka jälkeen tutustutaan myös ohjelmistontuotantoon, jonka merkitys on suuri tässä työssä. Seuraavaksi esitellään sulautetun järjestelmän toimintaa ja tutustutaan siihen, mitä se tarkoittaa ohjelmistonkehityksen näkökulmasta. Ennen varsinaista työn toteutusosaa esitellään IO-alusta, jonka kehitykseen tuleva kehitysympäristö suunnitellaan. Lopuksi katsotaan työn tulokset, analysoidaan hiukan tuloksia ja mahdollisia jatkokehitysideoita.

2 Ohjelmistoarkkitehtuuri

Ohjelmistojen koon suuretessa ja monimutkaistuessa ohjelmistoarkkitehtuurin merkitys kasvaa. Suuren ohjelmiston hallitsemiseksi kunnolla tehty arkkitehtuurisuunnittelu helpottaa koko ohjelmistonkehitysprosessia. Suunnittelutyön kannalta tärkeää on dokumentointi ja ohjelmiston jakaminen itsenäisiin osiin. Kun ohjelmointityön jakaa useampaan osaan, se helpottaa testaamista, ylläpitoa ja päivittämistä. Dokumentti arkkitehtuurista antaa kaikille yhteisen ymmärryksen ohjelmiston kehittämisestä ja sen rakenteesta. Suunnitteluvaiheessa olisi ensiarvoisen tärkeää ottaa huomioon ylläpito ja uudelleenkäytettävyysvaatimukset, koska se lisää työtehokkuutta ja valmista ohjelmakoodia voidaan hyödyntää tulevaisuudessa. Ohjelmiston arkkitehtuuri toimii järjestelmälle ohjenuorana ja yleisenä suunnitelmana, jossa kerrotaan suurin piirtein, kuinka asioiden tulisi toimia.

2.1 Ohjelmistoarkkitehtuurin määritelmä

Ohjelmistoarkkitehtuureille on useita erilaisia määritelmiä, ja usein arkkitehtuurista puhuttaessa sillä tarkoitetaan eri asiaa [4]. Arkkitehtuureille on kuitenkin olemassa IEEE-standardointi, jossa se määrittelee ohjelmistoarkkitehtuurin järjestelmän perusorganisaatioksi, joka sisältää järjestelmän osat, niiden suhteet ja suhteiden vaikutukset ympäristöön sekä periaatteet, jotka määrittävät järjestelmän suunnittelua ja evoluutiota. [3, s. 18.]

Arkkitehtuuri ei siis kata vain järjestelmän osittamista vaan myös järjestelmien väliset suhteet, eli arkkitehtuurin tehtävänä on ottaa kantaa ohjelmiston suunnittelun ratkaisuihin, jotka koskevat osiin jakamista ja niiden välistä kommunikointia, prosesseja, tiedon saantitapoja ja pysyvyyden toteuttamista. Voidaankin sanoa, että arkkitehtuuri on järjestelmän perustuslaki, jota on noudatettava järjestelmää suunniteltaessa ja sitä voidaan muuttaa vain todella hyvillä perusteilla. [3, s. 19.]

Siitä huolimatta, että ohjelmistoarkkitehtuureilla on useita määritelmiä, on sen käyttötarkoitus ja periaate kaikissa sama. Arkkitehtuuri toimii järjestelmän ytimenä, joka pysyy olennaisesti samana kehityksen ja ylläpidon aikana, ja sen mukaan voidaan suunnitella ja toteuttaa ohjelmisto.

2.2 Arkkitehtuuri ohjelmistonkehityksessä

Ohjelmistoarkkitehtuurit ovat muodostuneet omaksi ohjelmistotekniikan alueekseen verrattain myöhään, vasta 1990-luvun lopulla. Ohjelmistokehityksen alussa sovellukset pyrittiin näkemään muistipaikkoihin ja rekistereihin tallennettujen lukujen ja merkkijonojen käsittelynä. Sitten opittiin tunnistamaan ohjelmistojen muodostamia rakenteita, joita toteuttamaan kehitettiin kieliin rakenteisia tietotyyppejä. Vastaavasti tunnistettiin toimenpiteitä, joita näille rakenteille tuli suorittaa. Nämä esitettiin kielissä aliohjelmina. Myöhemmin opittiin myös näkemään sovelluksissa suurempia käsitteitä, joihin liittyi sekä tietoa että käyttäytymistä. Näitä toteuttamaan kehitettiin kieliin tietoabstraktio eri muodoissa. [3, s. 15.]

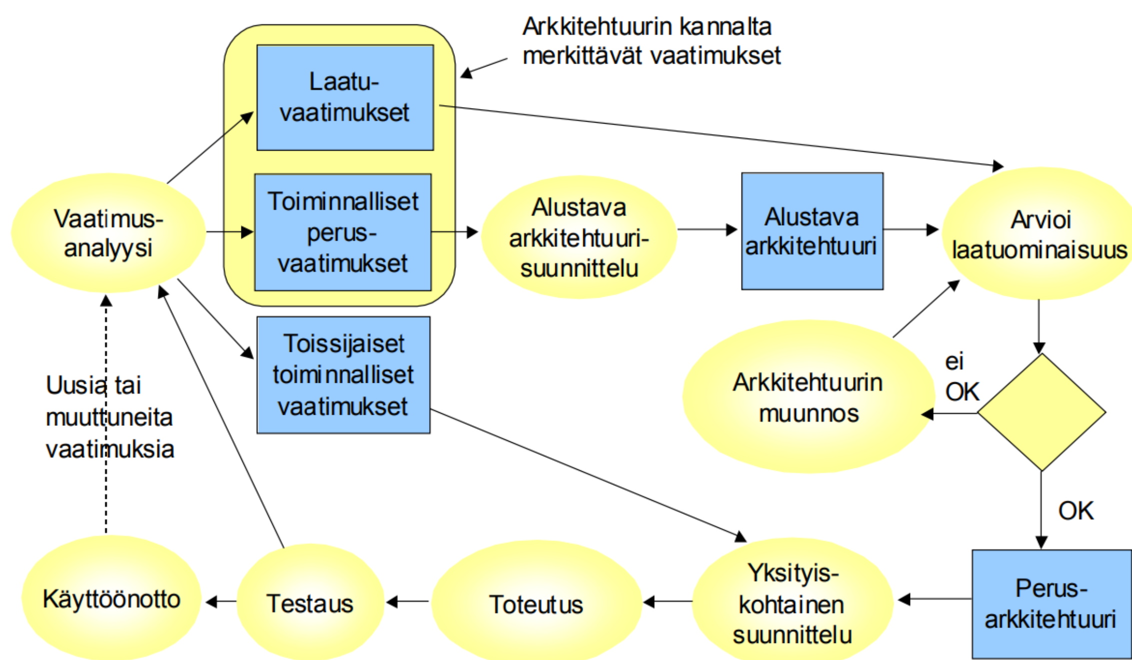
Tietoabstraktio kokoaa yhteen tietorakenteita ja niihin liittyviä operaatioita, jolloin tietoa käsitellään vain näiden operaatioiden kautta. Yksi tietoabstraktion muoto on olioparadigman tuoma luokkakäsite, jossa abstraktiin tietotyyppiin on yhdistetty laajennettavuus eli periytyminen. Huomattiin myös tarve koota suurempia kokonaisuuksia loogisesti yhteen kuuluvista palveluista. Nämä esitettiin tietyt rajapinnat toteuttavina komponentteina. [3, s. 16.]

Vähitellen yksittäisen sovelluksen käsite muuttui, eri sovellukset jakoivat yhteisiä komponentteja ja kirjastoja. Ymmärrettiin myös, että monilla eri sovellusalueille tarkoitetuilla ohjelmistoilla on olennaisesti sama perusarkkitehtuuri, jolloin kehitettiin ohjelmistoalustoja (engl. platform), jotka toteuttavat arkkitehtuurin tarvitseman yleisen infrastruktuurin. [3, s. 16.]

Ohjelmistojen luominen tapahtuu siis yhä useammin arkkitehtuurikäsitteiden mukaisesti. Tähän on vaikuttanut järjestelmien jatkuva monimutkaistuminen ja kasvaminen, myös uudelleenkäytön merkityksen kasvaminen ja teknologian kehitys. Arkkitehtuurin merkitys näkyy siis monessa asiassa ohjelmistonkehitysprosessissa. [3, s. 16.]

Hyvä arkkitehtuuri siis jakaa ohjelmiston moneen eri palaseen, jolloin se helpottaa esimerkiksi testausta ja ylläpitoa. Voitaisiin sanoa, että arkkitehtuuri antaa abstraktiotason näkymän itse ohjelmistoon, mikä mahdollistaa monimutkaisten ohjelmistojen suunnittelun ja josta käy ilmi erillisten komponenttien toiminta ja niiden väliset rajapinnat. Arkkitehtuurien arvoa kasvattaa myös se, että arkkitehtuuri on kunnolla määritelty ja dokumentoitu, jolloin saadaan mahdollisista kriittisistä tekijöistä tieto jo hyvin varhaisessa vaiheessa, jolloin koko järjestelmän muuttaminen on vielä edullista. [3, s. 17.]

Järjestelmän epäonnistuessa voidaan monesti siitä syyttää huonoa arkkitehtuuria, koska yleensä huono arkkitehtuuri ilmenee kehittämisen, käyttämisen tai ylläpidon aikana. Pahimmassa tapauksessa huonon arkkitehtuurin takia huomataan kesken suunnittelutyön, että koko järjestelmää ei voida toteuttaa. Arkkitehtuuri voi myös tehdä testauksesta ja ylläpidosta todella kallista ja vaikeaa, koska suunnittelutyössä näitä ei ole tarpeeksi huomioitu. [3, s. 17.] Esimerkki arkkitehtuuriperustaisesta ohjelmistonkehitysprosessista on esitelty kuvassa 1.



Kuva 1. Arkkitehtuuriperustainen ohjelmistokehitysprosessi [3]

Vaikka arkkitehtuurisuunnittelussa on kyse kokoelmasta yhteensopivia ratkaisuja, joilla pyritään täyttämään kaikki vaatimukset, toteutuksista huomataan usein, että tietty komponentti on ollut muita dominoivammassa tilassa arkkitehtuuria suunniteltaessa. Tyypillisesti jokin vaatimuksista on arkkitehtuurillisesti merkittävin, jolloin päästään tiettyyn perusarkkitehtuuriin, johon muut vaatimukset pakotetaan. Yksi arkkitehtuurin olennaisista tehtävistä onkin kuvata, kuinka arkkitehtuurisesti merkittävät vaatimukset täyttyvät annetulla arkkitehtuurilla. Tässä mielessä arkkitehtuuri nähdään kokoelmana korkeamman tason suunnitteluratkaisuja, jotka pyrkivät laadullisten vaatimusten ja niistä syntyvien ongelmien ratkaisuun. [3, s. 22.]

3 Ohjelmistotuotanto

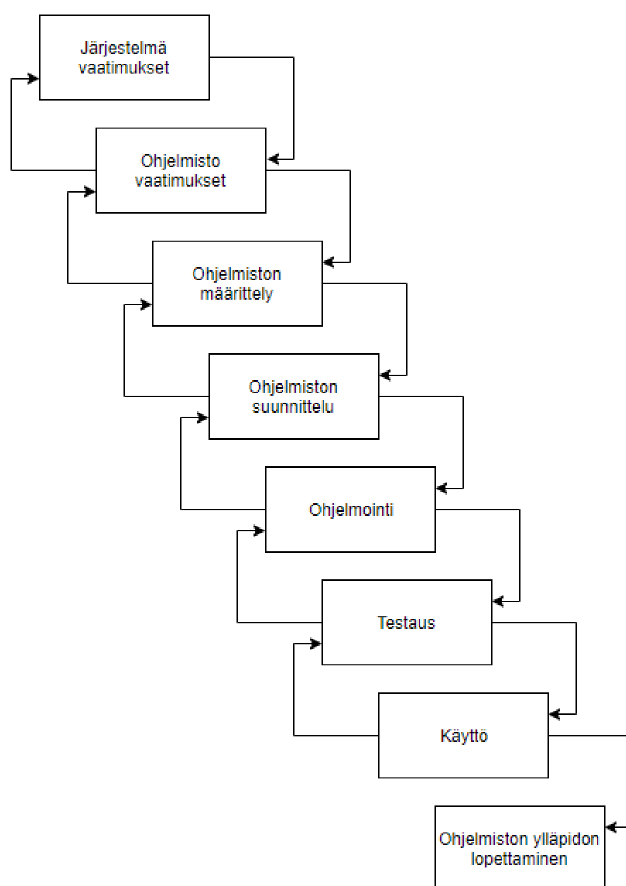
Ohjelmistontuotanto käsitteenä tarkoittaa koko ohjelmistonkehitysprosessia, eli se sisältää koko ohjelmistoprojektinhallinnan. Kaupallista ohjelmistontuotantoa tehdään yleensä projektityönä yhden tai useamman hengen tiimeissä.

Ohjelmistontuotannossa projekteissa on otettava huomioon kustannukset, aika, laatu ja ominaisuudet. Eli kuinka pystytään tekemään monimutkaisia järjestelmiä kohtuullisin kustannuksin, annetussa ajassa ja niin, että ne ovat kehitettävissä ja ylläpidettävissä tulevaisuudessa. Kustannukset ovat yleensä näistä rajoitetuin tekijä ja loppujen lopuksi kustannukset vaikuttavat kaikkiin muihin.

3.1 Ohjelmiston elinkaari

Ohjelmistonkehityksessä pyritään hakemaan ratkaisuja johonkin tiettyyn ongelmaan. Määritelmänä ohjelmistonkehitystä voitaisiin kutsua prosessiksi, jossa otetaan joukko vaatimuksia tai ongelmia, analysoidaan niitä, suunnitellaan ratkaisu niihin, ja implementoidaan ratkaisu tietokoneella. Ohjelmistokehitysprosessille ja siinä käytettäville menetelmille on useita vaihtoehtoja, käytännössä rajattomasti. Yksikään ratkaisu ei ole yleispätevä, ja ratkaisu, joka toimii yhdessä tilanteessa, ei välttämättä toimi toisessa tilanteessa ollenkaan. [5, s. 3.]

Jokaisella ohjelmistolla on myös oma elinkaarensa, sillä tarkoitetaan aikaväliä ohjelmiston kehittämisen aloituksesta sen poistamiseen käytöstä. Ei ole väliä, kuinka iso ohjelmisto on tai kuinka iso ryhmä sitä on tehnyt, kaikki ohjelmistot käyvät läpi samanlaisen elinkaaren. Ohjelmistonkehitystyö voidaan suorittaa monella eri tavalla. Näitä tapoja kutsutaan vaihejakomalleiksi. Vaihejakomallin tarkoituksena on jakaa ohjelmistonkehitystyö ja ohjelmiston elinkaari omiin vaiheisiinsa. Yksi tunnetuimpia vaihejakomalleja on vesiputousmalli, jonka yksi variaatio on esitelty kuvassa 2.



Kuva 2. Esimerkki vesiputousmallista

Vesiputousmallista näkee, minkälaisen elinkaaren ohjelmisto käy läpi. Vesiputousmallin on luonut Winston Royce, 1970-luvulla. Malli on hyvä esimerkki ohjelmistonkehitysmenetelmästä ja siinä tulee selkeästi esille ohjelmiston elinkaari, mutta nykyään ohjelmistonkehitysmenetelmiä on tusinoittain ja vesiputousmalliakaan harvemmin hyödynnetään niin kuin se oli 70-luvulla, vaan siihen on lisätty mukaan useita iteroimisvaiheita, kuten yllä olevassa variaatiossa. Muut ohjelmistonkehitysmenetelmät käyvät läpi aivan saman elinkaaren, mutta toistavat prosessia eri tavalla. Tänä päivänä suosituimpia ohjelmistonkehitysmenetelmiä ovat niin sanotut ketterät menetelmät, jolloin pystytään muuttamaan projektin kulkua ja suuntaa hallitusti kesken kehitysprosessin.

Kaikissa elinkaaren vaiheissa on omat toimenpiteensä. Vaiheiden sisältö on karkeasti esitelty alla:

Esitutkimukset

Esitutkimusvaiheessa selvitetään järjestelmän vaatimukset, eli selvitetään asiakastarpeet ja se, miksi tällainen järjestelmä tehdään ja mitä sen täytyisi pystyä tekemään. Esitutkimuksen tarkoituksena on vastata kysymykseen, miksi tehdään, ei muuhun.

Määrittely

Määrittelyvaiheessa mietitään ohjelmiston vaatimuksia ja määritellään, mitä ohjelmiston tulisi pitää sisällään. Määrittelyn tuloksena saadaan aikaan niin sanottu toiminnallinen määrittely. Toiminnallisen määrittelyn on tarkoituksena kuvata ohjelmiston toiminnot ja toteutukselle asetettavat ei-toiminnalliset vaatimukset.

Suunnittelu

Suunnitteluvaiheessa suunnitellaan määrittelyn mukainen toimintojen toteutus. Yleensä suunnittelu jaetaan kahteen eri osaan. Alussa suoritetaan arkkitehtuurisuunnittelu, jossa kokonaisuus jaetaan itsenäisiin osiin, niin sanottuihin moduuleihin. Seuraavassa vaiheessa suunnitellaan näiden moduulien sisältö.

Toteutus

Toteutusvaiheessa ohjelmoidaan ja toteutetaan suunnitelman mukainen ohjelmisto.

Testaus

Testausvaiheessa ohjelmistoa testataan ja pyritään poistamaan mahdolliset virheet.

Käyttöönotto

Käyttöönotto tarkoittaa ohjelmiston ylläpitämistä, jolloin ratkotaan asiakkaan ongelmia ja korjataan uusia mahdollisia virheitä ohjelmistossa.

Ylläpidon lopettaminen

Periaatteessa ohjelmisto ei kuole koskaan, mutta jossain vaiheessa sen ylläpito lopetetaan. Ohjelmistoa voi siis käyttää, mutta sitä ei enää korjata tai päivitetä ainakaan valmistajan toimesta.

Yleensä kaikkiin näihin vaiheisiin liittyy laadunvarmistustoimenpiteitä, jolloin elinkaaressa saatetaan palata askelia takaisin päin, tehdäkseen tarkastuksia, katselmuksia ja testausta. Tätä kutsutaan iteroimiseksi.

3.2 Ohjelmiston ominaisuuksia

Ohjelmistot voidaan karkeasti jakaa ominaisuuksiensa perusteella omiin kategorioihinsa. Tällaisten kategorioiden avulla voidaan luonnehtia ohjelmistoa periaatteellisella tasolla, mikä helpottaa ymmärtämään ohjelmistonkehitystyön tärkeimpiä haasteita. Esimerkkinä muutamia olennaisia asioita, joita yleensä jokainen ohjelmistonkehittäjä joutuu miettimään kehitystöitä suunniteltaessa:

- Suunnitellun ohjelmiston suuruus: Suuremmat ohjelmistot tarvitsevat kehitysvaiheessa paljon työntekijöitä ja tukea tuotantoprosessilta. Pienet ohjelmistot voidaan suunnitella jopa yhden kehittäjän voimin, jolloin tuen ja suunnittelun määrä on huomattavasti pienempi. Yleisesti ohjelmiston kokoa voidaan mitata sen ominaisuuksien määrällä tai ohjelman käsittelemän tiedon määrällä. [5, s. 12.]

- Reaaliaikaisuus: Järjestelmät, joissa toimintojen ajoittaminen on kriittinen tekijä, aiheuttaa suunnittelutyöhön haasteita. [5, s. 12.]

- Hajauttaminen ja sulautus: Järjestelmien hajauttaminen ohjelmistoissa tuottaa aina lisähaasteita suunnittelussa, koska useamman komponentin täytyy toimia rinnakkain, jolloin ohjelman suoritusjärjestykseen voi tulla ongelmia. Sulautetuissa järjestelmissä tärkeänä tekijänä on erottaa kohta, missä laitteiston ominaisuudet loppuvat ja ohjelmiston alkavat. Sulautetussa järjestelmässä myös ohjelmistojen toteuttaminen on paljon monimutkaisempaa. [5, s. 13.]

- Luotettavuus: Kriittisissä paikoissa toimivien ohjelmien on kyettävä toipumaan ongelmatilanteista ja toimimaan mahdollisimman häiriöttömästi, jolloin suunnittelussa on kiinnitettävä huomiota itse järjestelmän laatuun ja mahdollisiin varajärjestelmiin. [5, s. 13.]

- Skaalautuvuus: Kun järjestelmää käytetään useissa eri ympäristöissä, joissa on erilaiset kokoonpanot ja joiden suorituskyky vaihtelee, on suunnittelutyöhön panostettava paljon enemmän kuin sellaiseen järjestelmään, joka on vain yhteen kokoonpanoon. [5, s. 13.]

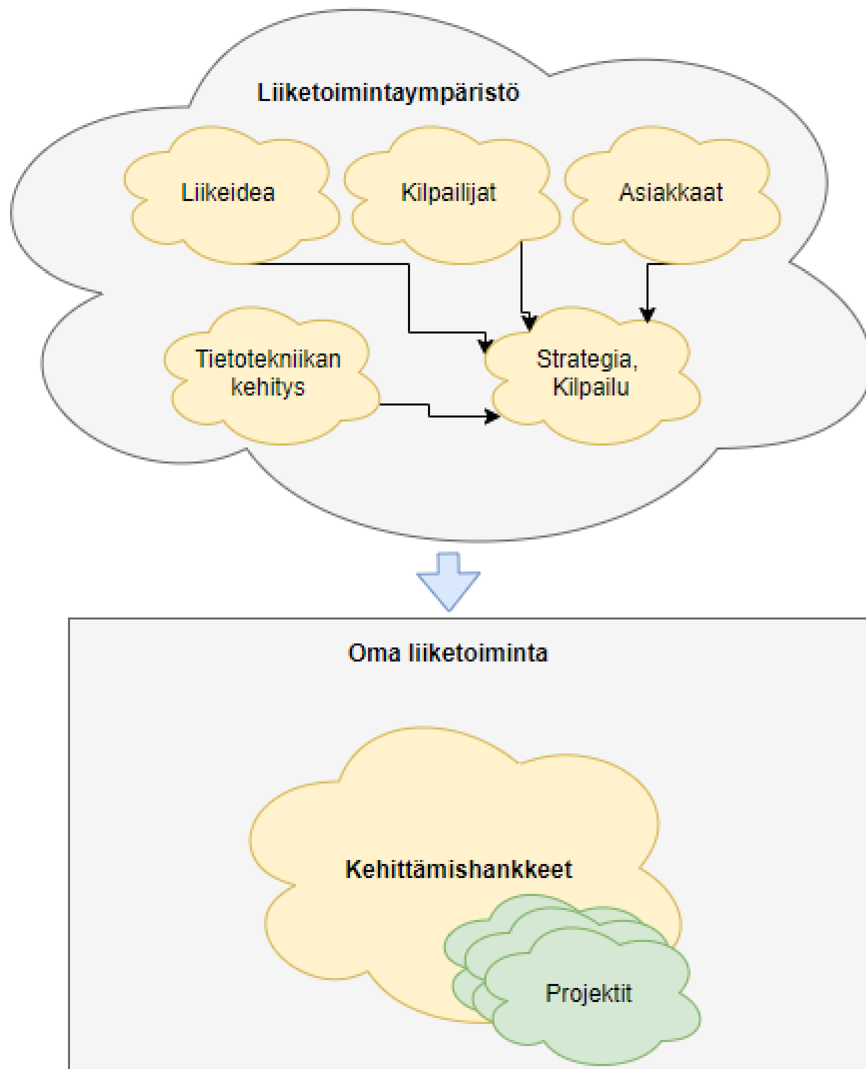
- Tuotteistus: Suunniteltaessa ohjelmaa, joka pohjautuu jo johonkin olemassa olevaan, testattuun järjestelmään, voidaan toteutusvaiheessa selvitä huomattavasti paljon pienemmällä työllä. Syynä tähän on, että yleensä vakiintuneisiin ympäristöihin on saatavilla tarvittavat komponentit ja työkalut valmiina ja myös tällöin oppimiskynnys on huomattavasti matalampi kuin uusiin ympäristöihin tutustuessa. [5, s. 13.]

3.3 Ohjelmistonkehitys projektina

Tyypillisesti ohjelmistonkehitystyöt organisoidaan projekteiksi, jolloin projekteissa toteutetaan asiakkaalta saatujen vaatimuksien perusteella haluttu ohjelmisto. Yleensä ohjelmistonkehitysideat perustuvat asiakkaan liiketoiminnallisiin tavoitteisiin, joilla pyritään tehostamaan asiakkaan toimintaa. Ohjelmiston toimittaja siis pyrkii ratkaisemaan asiakkaan ongelman tietoteknisin apuvälinein, jolloin asiakkaan on kommunikoitava toimittajan kanssa selkeästi siitä, mikä ongelman juurisyy on. Yleensä ohjelmistonkehitysprojekteissa puhutaan ongelman ratkaisuista, mutta monesti taustalla voi olla jokin kehitystyö, jolloin halutaan kehittää esimerkiksi jonkin olemassa olevan laitteen ominaisuuksia. [5, s. 19.]

3.3.1 Asiakkaan kannalta

Asiakkaan kannalta ohjelmistoprojektin tavoitteet ovat yleensä puhtaasti liiketoiminnalliset, jolloin asiakas näkee itse projektin paljon laajempana kokonaisuutena. Koko projektilla pyritään kehittämään yrityksen toimintaa, ja projekti voi olla osa jotain suurempaa hanketta. Tällaisilla hankkeilla on selkeä elinkaari, idea tuotantoon ja loppujen lopuksi tuotantokäytöstä poisto. [5, s. 19.] Liiketoimintaa havainnollistava periaate on esitelty kuvassa 3.

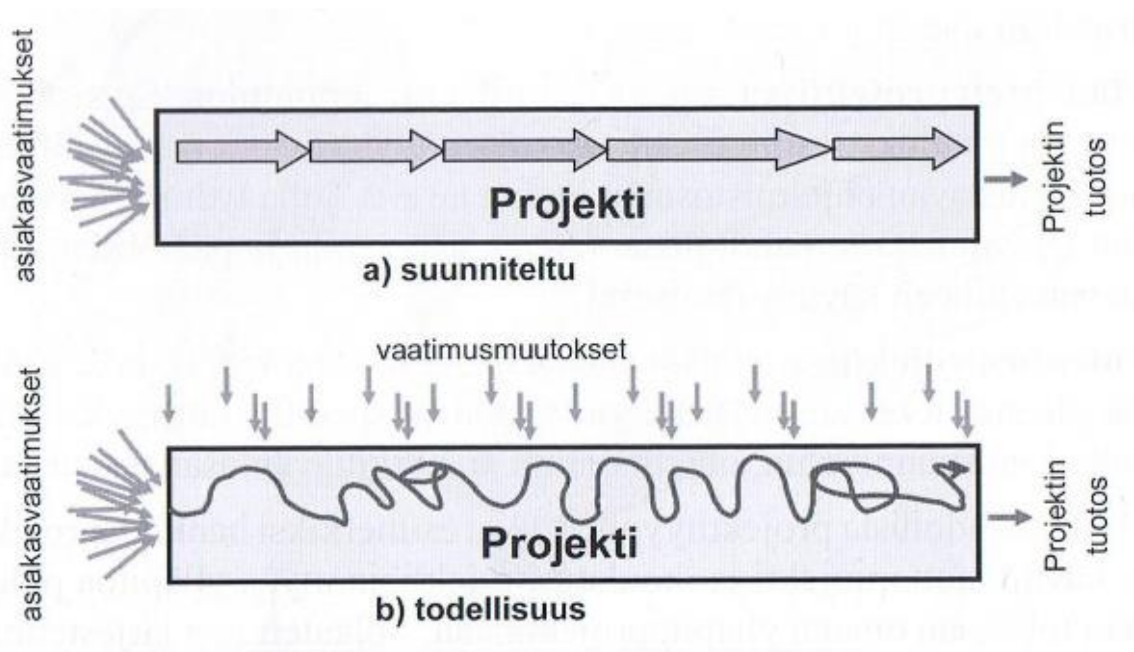


Kuva 3. Liiketoiminta (mukailtu lähteestä 5)

Tällaiset hankkeet sisältävät yleensä useita projekteja, kuten esitutkimus, määrittely ja käyttöönotto. Yhteistä hankkeen kaikille projekteille on se, että niillä on selkeä sisältö ja tavoite. Projekteille annetaan omat resurssit, kuten työhön osallistuvien työtekijöiden määrä ja alihankintaan käytettävän rahan määrä tai jotain muuta, jonka avulla projekti saadaan suoritettua. Rahan eli kustannuksien lisäksi toinen kriittinen tekijä projekteissa yleensä on aikataulu, jonka puitteissa tulisi projekti olla tehty. Pahimmassa tapauksessa projektiviivästykset voivat viivästyttää koko hankkeen valmistumista. [5, s. 20.]

3.3.2 Toimittajan kannalta

Ohjelmiston toimittajan kannalta lähtökohtana toimivat vaatimukset, joiden mukaan ohjelmistoa tulisi lähteä suunnittelemaan. Periaatteessa vaatimukset pitäisi kuvailla mahdollisimman tarkasti ja yksityiskohtaisesti, mutta todellisuudessa tilanne on harvoin näin. Yleensä vaatimuksia on mahdotonta kuvailla todella tarkasti, koska asiakas ei välttämättä projektin alkaessa itsekään täysin niitä tiedä. Projektin edetessä vaatimukset myös muuttuvat ja niihin saattaa tulla lisäyksiä koko projektin ajan. [5, s. 21.] Projektin kulkemisen periaate on esitelty kuvassa 4, jossa kuvataan projektin kulku niin kuin se on suunniteltu, mutta todellisuudessa matka on paljon mutkaisempi.



Kuva 4. Projektin kulku [5]

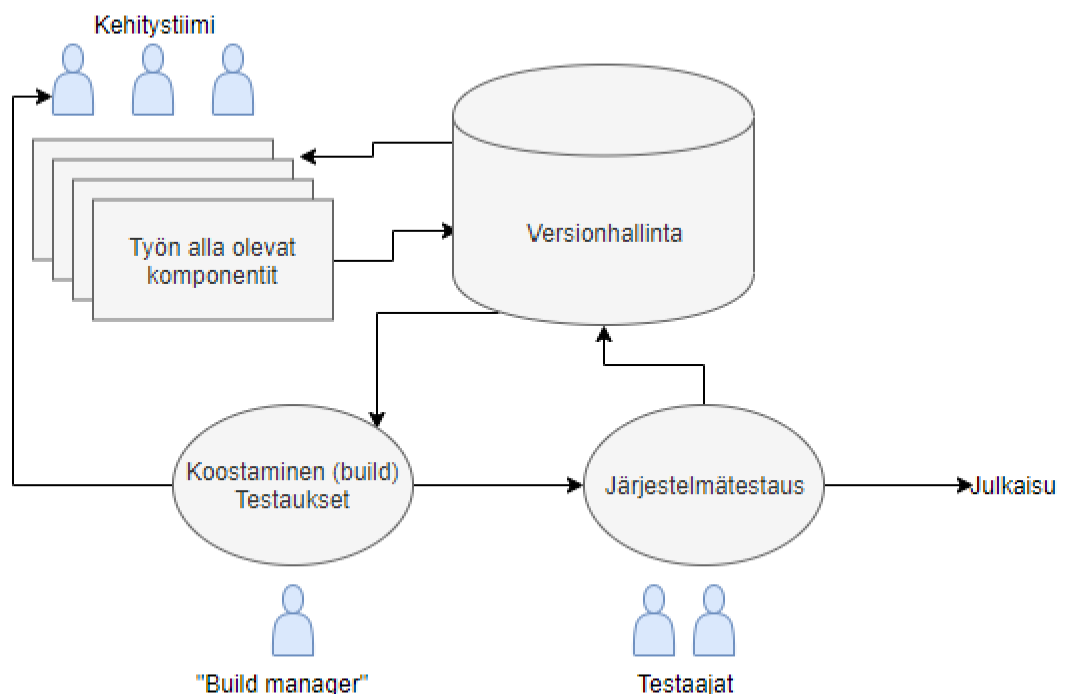
Tietysti muutokset ja uudet vaatimukset projektissa vaihtelevat asiakkaan ja ohjelman toimittajan sopimuksen mukaan, mutta kaikkiin muutoksiin loppujen lopuksi vaikuttaa aika ja kustannukset.

Nykyään ohjelmistotuotannossa käytetään paljon valmista, jolloin itse ohjelmisto perustuu johonkin ohjelmistoalustaan (engl. platform). Tällöin ohjelmisto perustuu johonkin jo olemassa olevaan tai samantapaiseen järjestelmään, joka on saatavilla toiseen ympäristöön tai sen alkuperäinen käyttötarkoitus on hiukan erilainen. [5, s. 23.]

Yksi ohjelmistotyön ongelmista on ohjelmistojen ylläpito, koska mitä useammalle asiakkaalle tehdään ohjelmistoja, tulee niitä myös ylläpitää. Tästä syystä tavoitteena olisi löytää ratkaisu niin, että yksi ohjelmistotuote palvelisi useaa asiakasta. Tämä yksinkertaistaisi ylläpitoa, kun kaikki tarvittavat muutokset voidaan tehdä samaan ohjelmistoon, ja esimerkiksi yhden asiakkaan muutoksesta voisivat hyötyä muutkin saman ohjelmiston käyttäjistä. Tämän esteenä on tietysti se, etteivät kaikki asiakkaat välttämättä halua oman ohjelmistonsa kehittyvän samaan suuntaan kuin toisen. [5, s. 23.]

3.3.3 Projektityön kulku

Projektissa työskentely vaatii hyvät työkalut, koska nopeat iteraatiot eivät onnistu ilman niitä. Normaalien ohjelmistonkehitystyökalujen lisäksi kehittäjä tarvitsee työkalut ainakin version- ja tuotteenhallintaan, koostamiseen, kommunikointiin, dokumentointiin ja testaukseen. [5, s. 56.] Kuvassa 5 esimerkki tuotekehityshankkeesta jatkuvan integroinnin tekniikalla.



Kuva 5. Esimerkki tuotekehitysprojektista (mukailtu lähteestä 5)

Esimerkissä kehitystiimin kehittäjät valitsevat tehtävälistasta jonkin tehtävän itselleen ja hakevat siihen liittyvät komponentit omaan kehitysympäristöönsä, jossa he voivat muokata ja testata sitä

vapaasti. Saatuaan oman versionsa toimimaan kehittäjä palauttaa muutetut komponentit versionhallintaan. Tässä vaiheessa on periaatteessa mahdollista, että versionhallinnassa tulee yhteentörmäyksiä, jos jostain syystä useampi kuin yksi kehittäjä on muokannut samoja komponentteja, mutta yleensä nämä saadaan kehittyneen versionhallinnan avulla ratkaistua. Integraatiopalvelimen havaittua uusien versioiden saapumisen versionhallintaan tehdään näistä komponenteista uusi kooste ja niille ajetaan automaattiset testit, jos sellaisia on, ja kooste menee onnistuneesti läpi. Tuloksena tästä syntyy raportti, josta käy ilmi onnistuiko vai epäonnistuiko kooste. Kaiken ollessa kunnossa menee kooste manuaaliseen testaukseen, jossa testaajat ajavat kokonaisen järjestelmän testaukset. [5, s. 57.]

Periaatteeltaan muutkin ohjelmistonkehitysmallit toteuttavat samaa kaavaa, mutta eri tavalla. Monesti kaikkiin näihin vaiheisiin liittyy vielä runsaasti dokumentointia ja raportointia työn etenemisestä ja tehdyistä muutoksista.

3.4 Ohjelmiston uudelleenkäyttö

Ohjelmistojen uudelleenkäytöllä voidaan ohjelmistojen tuottavuutta nostaa huimasti. Periaatteessa lähes mitä tahansa tuotteen vaihetta voidaan uudelleen käyttää, ja siksi uudelleenkäyttö on verrattain yleistä. Esimerkiksi jonkin järjestelmän määrittelyn pohjaksi voidaan ottaa edellisen samantapaisen järjestelmän määrittelystä syntynyt dokumentti. [5, s. 190.]

Yksi uudelleenkäytön muoto on ohjelman komponenttitasolla tapahtuva uudelleenkäyttö. Tällä tasolla uudelleenkäyttö on helpointa jakaa karkeasti kahteen eri luokkaan: yleiskäyttöisiin komponentteihin ja sovellusalue- ja sovelluskohtaisiin komponentteihin. Yleiskäyttöisiä komponentteja ovat esimerkiksi käyttöliittymien rakentamiseen tarkoitetut kirjastot, matematiikkakirjastot ja erilaiset tietorakennekirjastot. Sovellusaluekohtaiset komponentit liittyvät tietylle sovellusalueelle, esimerkkinä tietoliikenneverkkojen hallinta, joka voisi olla oma sovellusalueensa. Sovelluskohtaiset komponentit taas liittyvät tiettyyn sovellukseen tai tuoteperheeseen, esimerkiksi sovelluksen käyttöliittymäkirjasto. [5, s. 191.]

Uudelleenkäytön yhteydessä komponentit on usein muokattava käyttötarkoitukseensa sopivaksi, ja perinteisesti ongelma on ratkaistu muokkaamalla vanhaa versiota uuteen käyttöön sopivaksi. Tämä johtaa helposti tuotteenhallinnan painajaiseen, jossa muutoksien ja korjausten tekeminen kyseisen komponentin kaikkiin variaatioihin on erittäin suuri työ. Tietyissä tapauksissa muokkaaminen voidaan tehdä parametroimalla tai käyttämällä ehdollista kääntämistä. Komponenttitason

uudelleenkäytöllä tavoitellaan tuottavuuden ja laadun paranemista, koska ohjelmisto ei tarvitse tehdä kokonaan uudelleen. Muualla jo käytössä ollut komponentti on yleensä hyväksi havaittu ja todennäköisesti myös virheetön tai ainakin siihen liittyvät ongelmakohdat ovat jo tiedossa, jolloin testaukseen ja virheiden etsintään kuluvaa aikaa saadaan merkittävästi lyhennettyä. [5, s. 191.]

Käytännössä komponenttien uudelleenkäyttö ei ole niin yksinkertaista ja siihen liittyy joitain ongelmia, kuten kirjastojen luomisen ja ylläpitämisen vaativa työmäärä, komponenttien etsiminen ja dokumentoinnin puutteellisuus. On myös mahdollista, että komponentti on liian yleiskäyttöinen, jonka seurauksena komponentti saattaa olla hidas tai viedä liikaa muistia. Ohjelmakomponentit ovat myös usein niin monimutkaisia ja vaikeasti hahmotettavia, että tekijä saattaa kokea helpommaksi toteuttaa kokonaan uusi komponentti. [5, s. 192.]

Vaikka uudelleenkäyttö tarjoaisikin mahdollisuuden tuottavuuden nostamiseen, eivät uudelleenkäytettävät komponentit synny itsestään normaalin projektityön ohessa. Peukalosääntönä voidaan sanoa, että se vaatii ainakin 1,5-kertaisen työmäärän. Normaalissa projektityössä tästä muodostuu helposti ongelma, koska uudelleenkäytettävien komponenttien suunnittelu olisi suotavaa, mutta niihin annetut resurssit ja aika eivät riitä komponenttien toteuttamiseen. [5, s. 192.]

Yleisesti suunnitelmallinen ohjelmistojen uudelleenkäyttö perustuu aina arkkitehtuuritason ratkaisuihin. Esimerkiksi jos yritys toteuttaa paljon samantapaisia ohjelmistoja, voidaan niiden kehitys- ja ylläpitotyö yhdistää tuoteperheeksi, jolloin uudelleenkäytettävyysastetta saadaan huomattavasti kasvatettua. [5, s. 192.]

Kokonaisiin tuoteperheisiin liittyy myös omat ongelmansa. Esimerkiksi sovellusalue, jolla tuoteperhe toimii, tulisi tuntea tarkkaan, ja monesti yleinen ongelma on myös skaalautuvuus, sillä geneerinen tuoterunko on usein liian järeä ratkaisu kaikkein keveimmille ja vaatimattomimmille tuotteille. [5, s. 194.]

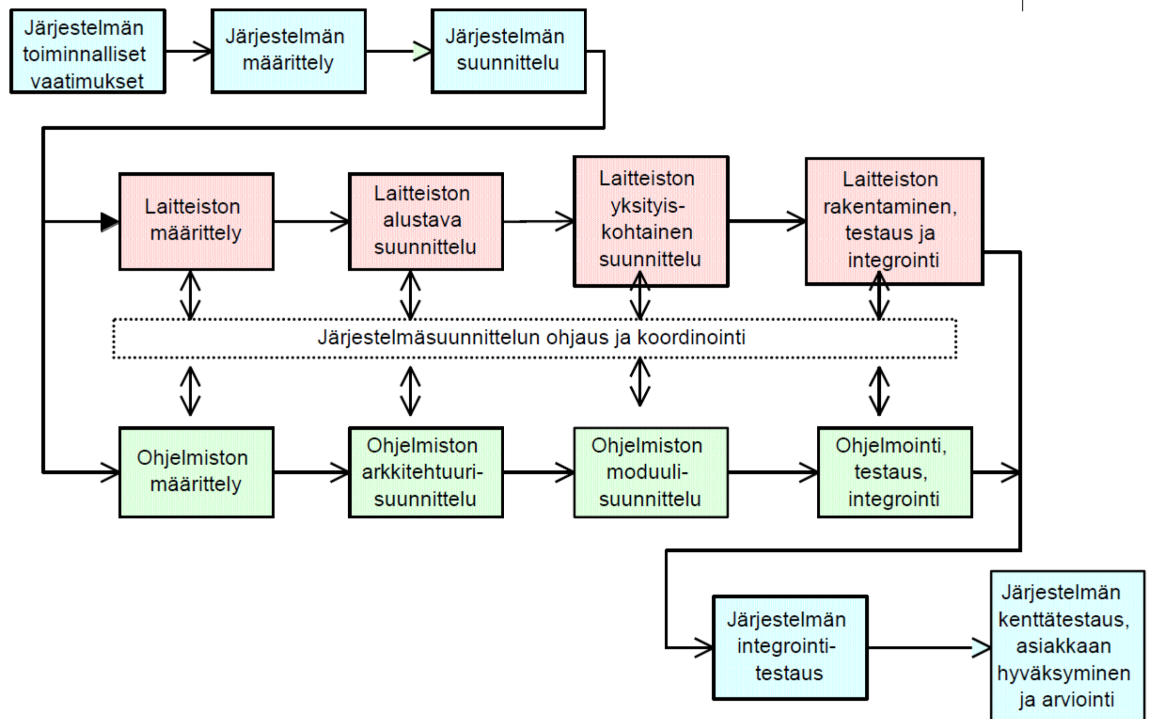
4 Sulautettu järjestelmä

Sulautettu järjestelmä on johonkin tehtävän suorittamiseen suunniteltu laite, jota ei välttämättä tunnista tietokoneeksi päältäpäin ollenkaan. Järjestelmät muodostavat suuren osan monesta arkipäivän laitteesta. Sulautetussa järjestelmässä ohjelmallinen osuus on sulautettu laitteen toimintaan siten, ettei laitteen käyttäjän tarvitse tiedostaa sen luonnetta ja olemassaoloa. Itse järjestelmä on kokonaisuutena paljon muutakin kuin tietotekniikka; paremminkin voitaisiin puhua mekatroniikasta eli kokonaisuudesta, jossa yhdistetään mekaniikkaa ja elektroniikkaa. [6, s. 10.]

4.1 Sulautetun järjestelmän suunnittelu

Sulautetun järjestelmän suunnittelu ja määrittely on monialainen tehtävä, koska koko järjestelmä voi sisältää useita eri tekniikan osa-alueen komponentteja. Tyypillisesti järjestelmä sisältää elektroniikkaa, ohjelmistoja ja mekaniikkaa. Tässä työssä perehdytään sulautetun järjestelmän elektroniikkaan suunnitteluun ja ohjelmistonsuunnitteluun.

Sulautettujen järjestelmien suunnittelutyössä elektroniikan ja ohjelmiston kannalta on omat haasteensa verrattuna esimerkiksi tavalliseen ohjelmiston suunnitteluun, koska järjestelmässä on kokonaisuutena useita huomioon otettavia asioita. Olennaista on huomata, että suunnittelun edetessä omine projekteineen laitteisto- ja ohjelmistopuolella on huolehdittava siitä, että rajapintojen epäselvyydet ja muutostarpeet huomioidaan tarpeeksi aikaisessa vaiheessa. Rajapinta- ja muiden yhteensopivuusongelmien ilmetessä vasta integrointivaiheessa työstä voi tulla kallis ja vaikea. Sulautetun järjestelmän suunnitteluesimerkki on esitelty kuvassa 6.



Kuva 6. Sulautetun järjestelmän kehitys [7]

Sulautetun järjestelmän koko voi vaihdella todella pienestä hyvinkin suureen kokoonpanoon, esimerkiksi sähköhammasharjasta isoihin teollisuusrobotteihin ja vaikka lentokoneisiin. [5, s. 14.]

Sulautetun järjestelmän toteutuksessa tulee ottaa huomioon useita eri tekijöitä, kuten virrankulutus ja laitteen toiminnan nopeus, eli kuinka voidaan varmistaa, että laitteet reagoivat tarpeeksi nopeasti, kun sisään tuleva tieto muuttuu tai kuinka paljon virtaa kuluu, kun laite suorittaa algoritmeja. Myös kommunikaatio ulkomaailman kanssa on otettava huomioon. [5, s. 14.]

Tyypillisesti muistin määrä ja hallinta on sulautetussa järjestelmässä haastava tekijä. Yleensä sulautetut ohjelmat vaativat vähän muistia, koska ohjelmat ovat pieniä, mutta todella pienissä systeemeissä ongelmia voi esiintyä, koska käytettävissä olevan muistin määrä on muutamia kilotavuja. Myös ohjelmistotyön toteutusympäristö on tärkeä tekijä. Yleensä työn alla olevan laitteen ohjelmointityössä käytetään sille räätälöityjä työkaluja, jotka voivat laadultaan olla erittäin heikkoja. Voi olla jopa niin, että projektissa joudutaan toteuttamaan omat kehitystyökalut, jotta ohjelmistotyö olisi mahdollista. Tästä syystä ohjelmistonkehitystyökalut ovat erittäin tärkeässä asemassa sulautettujen laitteiden kehitystyössä. [5, s. 14.]

Myös luotettavuusvaatimukset ovat raskaita sulautetuissa laitteissa, koska monesti ympäristöt, jossa laitteet toimivat ovat kriittisiä ja, jos laitteen sijainti on toisella puolella maapalloa saattaa

huollon järjestämisestä tulla erittäin kallista. Tämän takia näiden laitteiden tuotteistusaste pyritäänkin saamaan täydelliseksi, jolloin mahdolliset korjauskustannukset pysyisivät pieninä. Pahimmassa tapauksessa raskaat ylläpito- ja huoltokustannukset ylittävät helposti koko ohjelmiston tai laitteen kehityskustannukset.

4.2 Sulautettu elektroniikka

Sulautettujen järjestelmien elektroniikka on yleensä toteutettu hyvin pienien suorittimien avulla, mutta on myös mahdollista, että erittäin suurissa järjestelmissä joudutaan käyttämään hyvinkin tehokkaita suorittimia. Myös se minkälaisia toimintoja elektroniikka sisältää riippuu täysin järjestelmästä, mutta yleensä jokaisesta laitteesta löytyy ohjelmamuistia, datamuistia ja GPIO-nastoja. Tämän lisäksi mukana voi olla niin sanottuja oheislaitteita (engl. peripheral), kuten A/D-muuntimia, väylälogiikkaa, ajastimia, keskeytysohjaimia ja useita muita. Jokaiseen järjestelmään suunnitellaan siis omanlaisensa toteutus, joka palvelee juuri sen järjestelmän tarpeita.

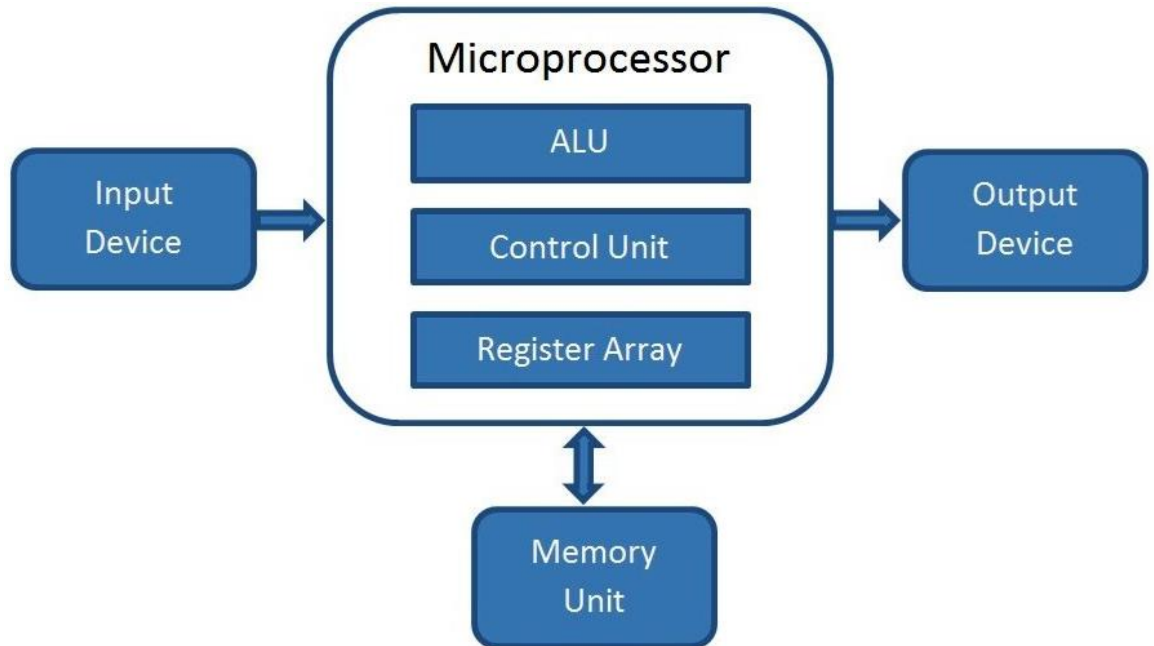
Sulautetun laitteen elektroniikkaa suunniteltaessa ja käytettävää piiriä valittaessa olisi hyvä ottaa huomioon myös skaalautuvuus. Eli mahdollisesti käyttää sellaista piiriä, joka toimisi mahdollisimman monessa tuotteessa. Skaalautuvuuden onnistuessa se on yrityksen kannalta edullisempaa, koska henkilökunnan ei tarvitse opetella useita erilaisia ympäristöjä projektien vaihtuessa ja samaa oppimista voidaan hyödyntää useiden eri projektien ja tuoteperheiden kanssa.

Elektroniikan käsitteet voivat olla hankalia ymmärtää ja useasti niitä sekoitetaankin, koska kunnollista standardointia ei ole. Yksi asia mikä monesti sulautetuissa laitteissa sekoitetaan, on käsite mikrokontrollerista, mikroprosessorista ja järjestelmäpiiristä (engl. system-on-chip). Seuraavissa olevissa aliluvuissa tarkastellaan näiden kolmen yleisen käsitteen merkitystä.

4.2.1 Mikroprosessori

Mikroprosessorilla viitataan suorittimeen tai prosessoriin, jossa kaikki suorittimen (engl. Central processing Unit) osat on pakattu yhdelle mikropiirille (engl. Integrated Circuit). Mikroprosessorin toiminta perustuu kellotaajuuteen ja rekistereihin, joka hyväksyy sisään tulevana tietona binääridataa. Data käsitellään muistiin tallennetun käskykannan mukaisesti, jonka jälkeen data syötetään piiristä ulos. [8.]

Datan käsittelyyn vaikuttaa se, mikä käskykanta on käytössä. Käskykanta muodostuu sen mukaan, mikä mikroprosessoriarkkitehtuuri on käytössä. Esimerkiksi useissa älypuhelimissa käytetään ARM-arkkitehtuurin mukaista käskykantaa, kun taas monessa kotitietokoneessa käytetään AMD x86 mukaista käskykantaa. Yleisimmät arkkitehtuurit ovat nykyään 32- tai 64-bittisiä. [8.] Kuvassa 7 on esitelty mikroprosessorin toimintaperiaate.



Kuva 7. Mikroprosessorin toiminta [9]

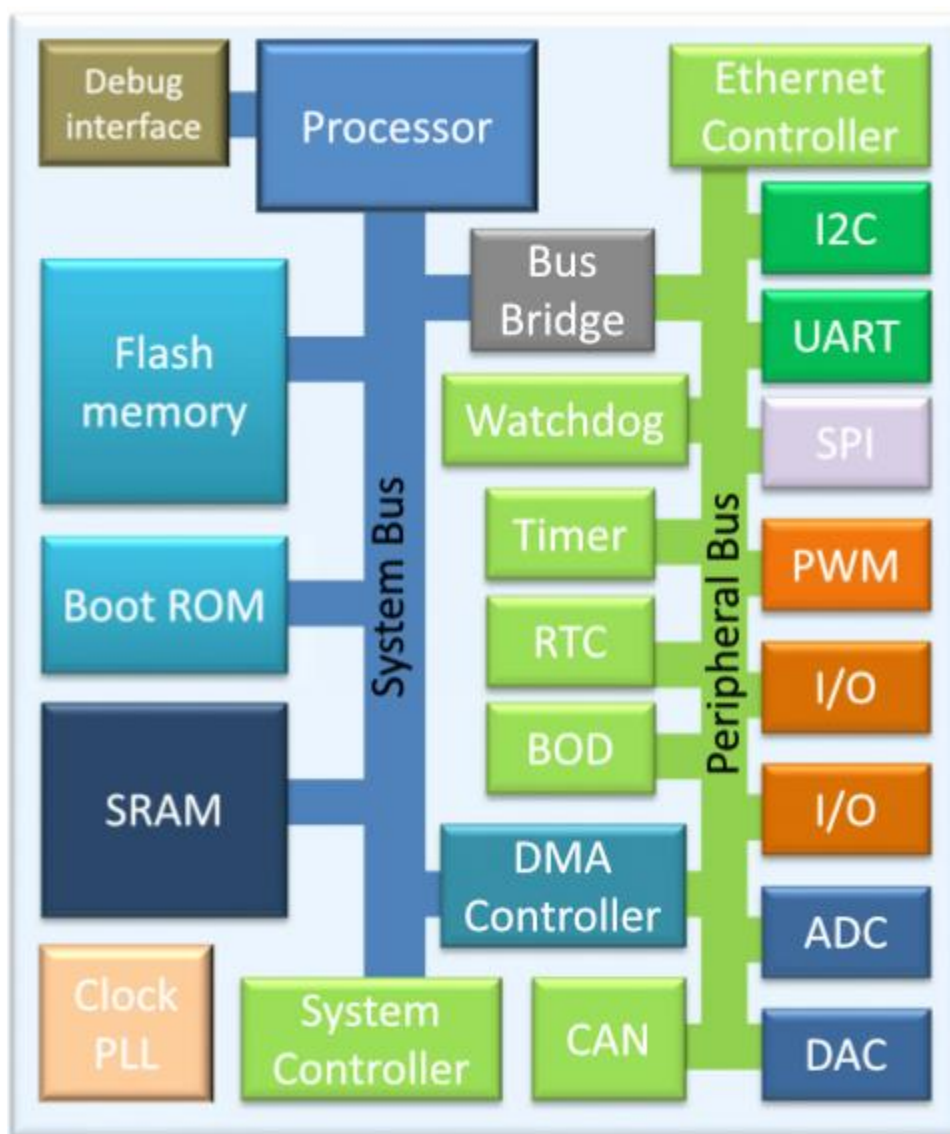
Mikroprosessori on siis laitteen aivot, joka koostuu pääasiassa kuvan mukaisista osista:
ALU on aritmeettinen ja looginen yksikkö, jonka tarkoitus on suorittaa kaikki aritmeettiset ja loogiset toiminnot sisään tulevalle datalle. Rekisteritaulukko (engl. Register array) koostuu useista rekistereistä, joiden tarkoitus on toimia väliaikaisena ja nopeana tallennustilana datan käsittelyä varten. Ohjausyksikkö (engl. Control unit) hoitaa käskykantaa ja datan liikuttamista koko järjestelmässä. [9.]

4.2.2 Mikrokontrolleri

Mikrokontrolleri on pieni tietokone, johon on integroitu mikropiiri. Mikrokontrolleri sisältää aina vähintään yhden mikroprosessorin, muistia ja ohjelmoitavia oheislaitteita (engl. peripheral). Yleensä mikrokontrollerit on suunniteltu käytettäväksi sulautettujen laitteiden kanssa, kun taas

pelkkiä mikroprosessoreita hyödynnetään esimerkiksi kotitietokoneissa. Monia kotitalouden laitteita ohjataankin mikrokontrollereiden avulla. [8.]

Kuvassa 8 on esitelty tyypillinen mikrokontrollerin toteutus. Kuvassa on prosessori, joka voisi olla esimerkiksi tässä työssä käytettävä ARM Cortex-M4, joka keskustelee väylien välityksellä muistien ja muiden oheislaitteiden kanssa. Kuvan toteutuksesta käy myös ilmi virheenkorjausrajapinta (engl. Debug interface), jonka kautta yleensä mikrokontrollerit ohjelmoidaan, joissakin toteutuksissa on myös mahdollista ohjelmoida mikrokontrolleri oheislaiteväyliä pitkin, kuten UART:n kautta.

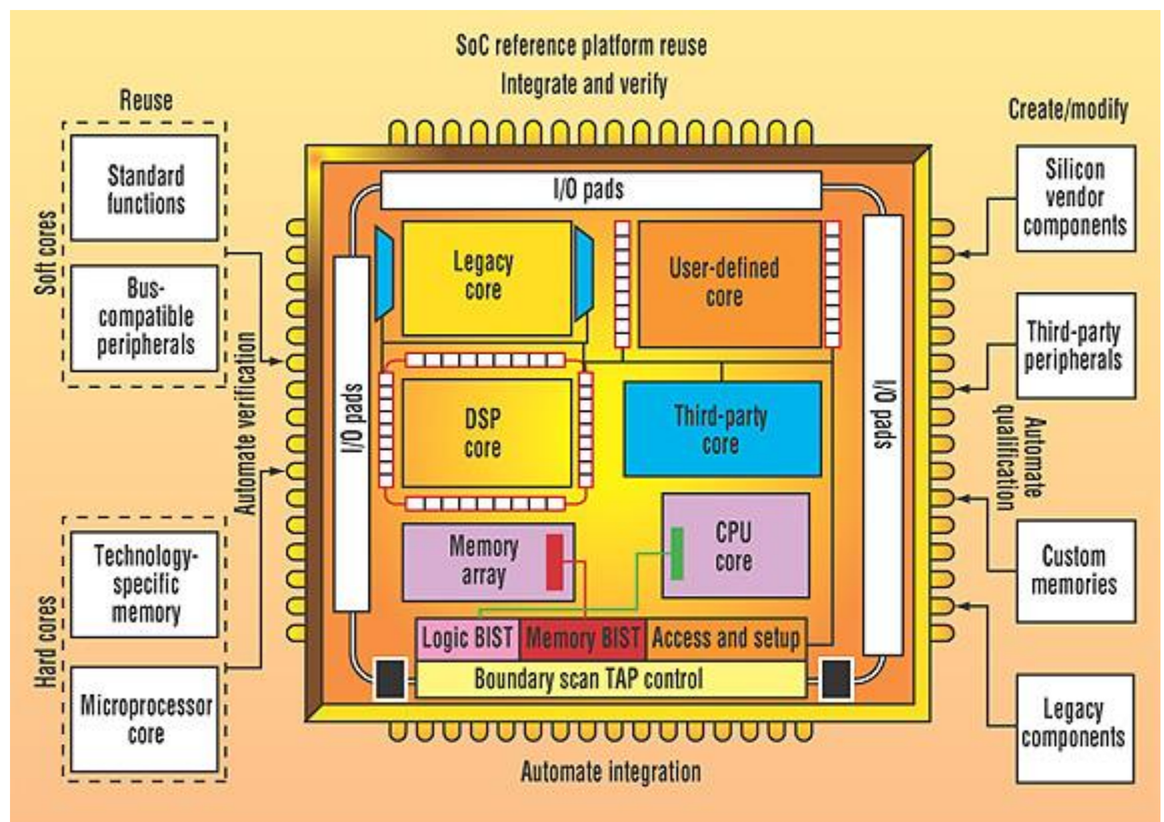


Kuva 8. Mikrokontrolleri [10]

Yleisesti voidaan sanoa, että mikrokontrollerin hyötyjä on helppo I/O:n ohjaus, toimintojen ajoituksista saadaan tarkkoja, käynnistyy välittömästi, virrankulutus on pieni ja se myös skaalautuu tarvittaessa pieneksi.

4.2.3 Järjestelmäpiiri

Elektroniikkaa suunniteltaessa esille nousee myös helposti kysymys, että käytetäänkö mikrokontrolleria vai niin sanottua järjestelmäpiiri tyyppistä (engl. System-on-Chip) ratkaisua. Järjestelmäpiiri sisältää käytännössä kaiken, mitä tietokoneen tarvitsee sisältää toimiakseen. Järjestelmäpiiriä voisi siis verrata kotitietokoneen emolevyyn. Emolevyyn vain lisätään haluttuja komponentteja itse, kuten näytönohjain, muistia, kiintolevyjä yms. Järjestelmäpiiriin nämä on jo itsessään integroitu. Yleensä järjestelmäpiirit rakennetaan mikrokontrollerin ympärille, joka taas on rakennettu mikroprosessorin ympärille. [8.] Kuvassa 9 on esitelty esimerkki järjestelmäpiiristä.



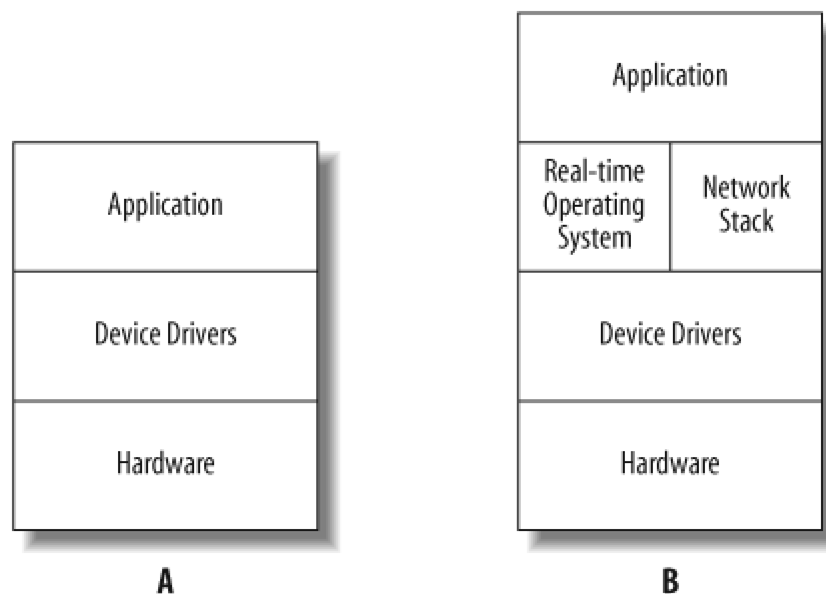
Kuva 9. System-on-Chip periaate [11]

Useat nykypäivän laitteet sisältävät järjestelmäpiirin, jossa käytetään ARM-mikroprosessoreita. Järjestelmäpiirin hyötynä on se, että yleensä ne ovat tehokkaampia kuin mikrokontrollerit, jolloin

ne pystyvät pyörittämään helposti esimerkiksi käyttöjärjestelmiä. Myös muistia on yleensä järjestelmäpiireissä enemmän, jolloin raskaat toiminnot toimivat paremmin [12]. Järjestelmäpiirejä hyödynnetäänkin useimmiten sellaisissa sovelluksissa, jotka vaativat pieneen tilaan tehokkaan laitteiston. Sulautettuja laitteita suunniteltaessa onkin mietittävä tarkasti minkälaista toteutusta kannattaa käyttää, eikä asia ole monesti yksinkertainen valittava.

4.3 Sulautettu ohjelmointi

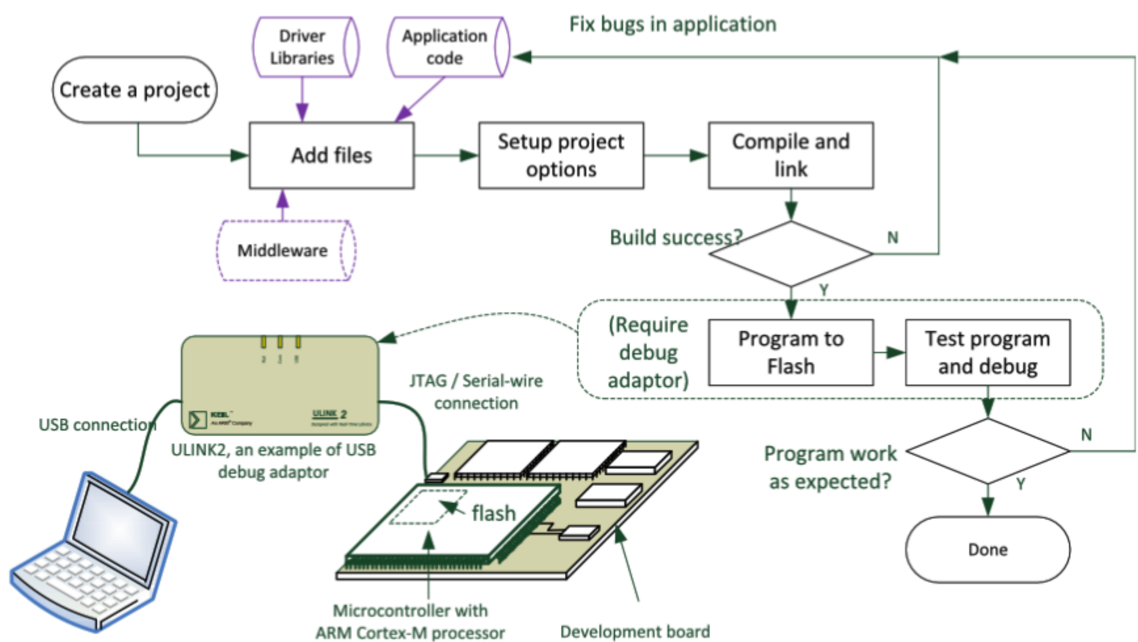
Sulautettujen järjestelmien ohjelmointi ei periaatteeltaan eroa muusta ohjelmoinnista. Se vain pitää sisällään joitakin erityispiirteitä, jotka on otettava huomioon järjestelmän suunnittelussa ja toteutuksessa. Esimerkiksi yksi erityispiirre on se, että sulautetun laitteen kehitystyö ja testaus suoritetaan eri ympäristössä kuin missä itse ohjelman lopullinen suoritus tapahtuu. [13, s. 148.] Kuvassa 10 on esitelty esimerkki, miten sulautettu ohjelmisto muodostuu.



Kuva 10. Yksinkertainen ja monimutkaisempi sulautettu ohjelmisto [14]

Kuvasta käy ilmi, kuinka sulautettu ohjelmisto muodostuu useasta eri lohkosta. Tällä on tarkoitus havainnoida sitä, että ohjelmistojen toiminnalliset kerrokset tulisi pitää erillään toisistaan. Tällä tavoin ohjelmistoa tehdessä kokonaisuus pysyy mahdollisimman siistinä, helposti luettavana ja helpommin siirrettävänä uuteen ympäristöön. [14.]

Monesti oman haasteensa tuo myös se, että järjestelmät on rakennettava käyttäen erikoislaatuisia työkaluja, jotka on suunniteltu juuri kyseiselle mikrokontrollerille tai prosessorille. Tämän takia onkin aiheellista testata myös kehitysympäristö, jolla laitteita tullaan kehittämään. Rajapintaerojen lisäksi sulautetuissa järjestelmissä on monesti sellaisia laitteita, joita ei kehitysympäristössä ole käytettävissä, jolloin testaus on suoritettava itse kohdelaitteessa. Testaus kohdelaitteessa saattaa olla työlästä, ja usein kehitystyövaiheessa se ei ole kovin järkevää. Jotta testausta kohdelaitteessa olisi mahdollisimman vähän, käytetään apuna usein testiympäristöjä. Testiympäristö on ohjelmisto, joka toimii kehityskoneessa. Tällä pyritään simuloimaan mahdollista kohdeympäristöä mahdollisimman tarkasti. [13, s. 148.] Kuvassa 11 on esitelty esimerkki sulautetun laitteen kehittämisestä.



Kuva 11. Sulautetun laitteen kehittäminen [10]

Periaatteessa kehitysympäristö saattaa olla hyvinkin monimutkainen, jos samaa ohjelmistoa käytetään useassa eri kohteessa. Itse ohjelmointiympäristönä voi toimia kehittäjän oma henkilökohtainen työkone ja ristiinkäännöskoneena jokin verkossa oleva toinen kone, josta se siirtyy kolmannelle koneelle, missä ohjelmistoa testataan. Monimutkaisissa ympäristöissä vaaditaankin kehittäjältä tarkkuutta ja itsekuria, jotta ohjelmistojen siirrettävyys ympäristöjen välillä olisi mahdollisimman helppoa. Tästä syystä ympäristökohtaiset vaatimukset tulisi minimoida. [13, s. 148.]

Sulautettujen laitteiden ohjelmoinnissa käytetään yleensä ohjelmointikielenä C:tä, C++:aa tai Javaa. Nykyään markkinoilla on myös laitteita, joita voi ohjelmoida Python-ohjelmointikielellä. Kai-

kista yleisin sulautetun laitteen ohjelmointikieli on kuitenkin C eli niin sanotusti ANSI C. Olipa pääohjelmointikieli mikä tahansa, tietyt järjestelmän osat vaativat Assembly-kielen käyttöä, varsinkin kaikkein suorituskykykriittisimmissä ja laitteistoläheisimmissä tilanteissa, kuten laitteita optimoitaessa. [13, s. 149.]

4.3.1 Kehitysympäristö

Kehitysympäristöllä tarkoitetaan laitteistoja ja ohjelmistoja, joiden avulla piirille tehdään ohjelmistot. Yleensä ohjelmat tehdään piirille niin sanotusti ristikehityksenä, eli ohjelmat tehdään eri ympäristössä, kuin missä ohjelmistoa tullaan suorittamaan. Tämän takia kehitysympäristön täytyy tukea ohjelmointityötä.

Kehitystyön kannalta tärkeää on työkalujen hyvä laatu ja mieluusti laaja käytettävyys, jos suinkin on mahdollista. Sulautettujen järjestelmien parissa työkalut on monesti rajattu hyvin tarkasti tiettyyn ympäristöön, jolloin samoja kehitystyökaluja ei voi käyttää kuin samaan piiriin kehitystyötä tekevät henkilöt. Tästä syystä skaalautuvuus olisi ensiarvoisen tärkeää. [13, s. 133.]

Tiettyjen piirien kehitysympäristöt kannattaa mahdollisuuksien mukaan jäädyttää sopivaksi aikaa ja tarvittaessa ostaa esimerkiksi tukipalveluita kehitysympäristön kehittäjältä, jotta mahdolliset ongelmat saadaan ratkaistua mahdollisimman nopeasti. Sen sijaan yleisessä käytössä olevilla kehitysympäristöillä ongelmat saadaan ratkaistua nopeasti laajan käyttäjäkunnan takia. [13, s. 133.]

4.3.2 Virheet

Sulautetuissa laitteissa virheiden etsiminen voi olla todella haastavaa, koska ohjelmaa ei välttämättä voida testata suoraan kehitysympäristössä. Tällä tarkoitetaan sitä, että normaalia virheen jäljitysohjelmaa (engl. debug) ei voi käyttää, eikä kunnollisia testausohjelmiakaan ole suunniteltu. Pienetkin muutokset ohjelmistossa saattavat aiheuttaa ongelmia ja vieläpä niin, että itse ongelman syytä on hankala paikantaa, koska monesti sulautettujen laitteiden kanssa ongelman juurisyy saattaa olla jossain ihan muualla kuin aluksi luullaan. Varsinkin testauksien alkuvaiheessa ongelmaksi muodostuu se, että ei olla ihan varmoja, testataanko laitteistoa vai ohjelmaa ja virheiden tullessa, onko virhe ohjelmassa vai laitteessa. [13, s. 150.]

Jos ohjelmasta löydetään virhe, sen paikantamisesta saattaa tulla todella työlästä varsinkin isoissa ja moniosaisissa ohjelmissa. Virhe saattaa syntyä aivan eri puolella ohjelmistoa kuin missä se havaitaan. Esimerkiksi monisäikeisessä ympäristössä alkuperäinen vika voi tullakin aivan jostain toisesta säikeestä ja vikaa etsitään aluksi väärästä paikasta, johon tuhrautuu huomattavan paljon aikaa. [13, s. 151.]

Monesti hankalimmat viat ovat ajoitukseen ja muistiin liittyviä. Ajoitusongelmista haastavaa tekee niiden ilmenemisfrekvenssi, jolloin itse vika on hankala paikantaa, koska se saattaa ilmetä vain satunnaisesti ja ennalta arvaamattomasti. Myös dynaaminen muisti voi aiheuttaa epämiellyttäviä ongelmia. Tässäkin syynä sen hankala paikannettavuus, koska esimerkiksi jäänneviittaukset tai roskaantumiset voivat ilmetä aivan eri kohdassa kuin itse ongelma on. Tästä syystä dynaamisen muistin käyttöä tulisi välttää sulautetuissa laitteissa. [13, s. 151.]

4.3.3 Testaus

Testauksen avulla pyritään siihen lopputulokseen, että mahdollisimman suuri osa virheistä löydetään ja korjataan. Testaustapoja voi olla useita ja loppujen lopuksi testaus riippuu täysin laitteesta ja käytettävissä olevista työkaluista. Periaatteessa koko järjestelmän toimintaa ei voida testata kuin vasta lopullisessa kohdelaitteessa, mutta ainakin osa testauksesta voidaan tehdä jo kehitysympäristössä.

Kehitysympäristössä testaamisesta on paljon hyötyä, koska testaukseen ja virheiden löytämiseen on käytettävissä monipuoliset työkalut ja työskentely on huomattavasti nopeampaa. Tämän lisäksi itse laitetta ei tarvitse siirrellä kohdeympäristöön. Tämä asia ei ole kuitenkaan itsestään selvyys, koska aina nämä kehitysympäristön työkalut eivät ole optimaaliset juuri tämän laitteen testaukseen. Tästä syystä kehitysympäristöllä on suuri merkitys myös testauksen kannalta. Kuitenkin kehitysympäristössä laitetta kannattaa testata, jos se on mahdollista, sillä sen avulla laitteen looginen toiminta saadaan toimimaan oikein. [13, s. 154.]

Kehitysympäristössä voi mahdollisesti olla myös simulaattori, jolla pystytään simuloimaan tulevaa kohdeympäristöä. Simuloinnissa ohjelma käännetään kohdelaitteen ymmärtämään muotoon, mutta ohjelmaa ei ajeta kohdeympäristössä vaan simulaatiossa, joka on kehitysympäristössä. Simulaattori siis tulkitsee kohdekoneen konekieltä. [13, s. 155.]

Simulaattoriin voidaan esimerkiksi kuvailla oheislaitteita, joita mahdollisessa kohdeympäristössä on. Tällä tavoin voidaan testata ja jäljittää virheitä myös oheislaitteita ohjaavista ohjelman osista. On simulaattorissa tietysti ongelmansa, ja se on itse se simulointi, koska simuloinnissa eivät välttämättä ilmene läheskään samat ongelmat, jotka kohdelaitteessa voivat ilmetä varsinkin, jos kohdelaite on monimutkainen kokonaisuus. Tällöin oikeaa ympäristöä on vaikea simuloida ja reaktiot, mitkä tapahtuisivat oikeassa ympäristössä jäävät simuloinnissa tapahtumatta. [13, s. 155.]

Simulaattorin avulla voidaan kuitenkin suorittaa järjestelmän perustestaus ennen lopullisen testauksen suorittamista kohdelaitteessa. Kohdelaitteessa testaaminen on kuitenkin paljon hitaampaa ja kalliimpaa kuin simuloimalla nopeasti peruslogiikan ja toiminnallisuuksien testaaminen. Markkinoilla on saatavilla valmiita simulaattoreita suorittimille, mutta voidaan se toteuttaa myös itse käyttämällä siihen tarkoitettuja työkaluja. [13, s. 155.]

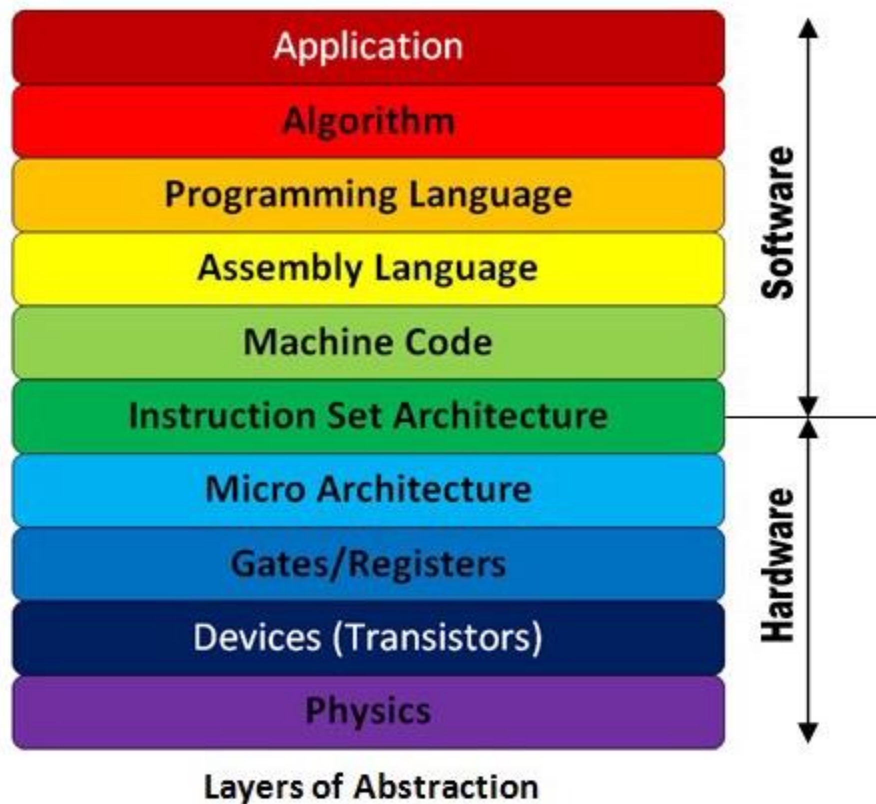
4.3.4 Ohjelmiston laitteistorajapinta

Laitteistoa ohjelmoitaessa toteutustapoja on monia, osa helpompia ja nopeampia, osa taas vaikeampia ja hitaampia. Esimerkiksi mikrokontrolleria ohjelmoitaessa voidaan ohjelmointi tehdä vaikka suoraan rekisteritasolla tai Assembly-kielellä, jolloin ohjelmointi tapahtuu niin sanotusti alimmalla tasolla. Tämä voi olla todella työlästä, mutta ohjelmoija saa tässä toteutuksessa yleensä tarkan tiedon siitä, mitä ohjelma tekee, kun kaikki käskyt täytyy toteuttaa itse. Yleensä ohjelmointi kuitenkin suoritetaan jollakin korkeamman tason ohjelmointikielellä kuten C-kielellä.

Prosessori voidaan jakaa kahteen tasoon, ylempään ja alempaan. Ylemmällä tasolla toteutetaan varsinaiset käyttöjärjestelmän ominaisuudet ja alemmalla toteutetaan liityntä laitteiston ja ohjelmiston välillä. Yleisesti tällaisesta laitteistoliittymärajapinnasta käytetään nimitystä Hardware Abstraction Layer (HAL). Tällaisen toteutuksen tarkoitus on se, että käyttöjärjestelmä voidaan vaihtaa helposti laitteesta toiseen vaihtamalla vain HAL-tason alapuolella olevaa toteutusta vastaamaan uutta laitteistoa. [13, s. 49.]

Myös käyttöjärjestelmäkohtaiset ominaisuudet voidaan joutua erottamaan toisistaan, jolloin voidaan hyödyntää niin sanottua Operating System Abstraction Layer (OSAL)-rajapintaa. OSAL-rajapinnan tarkoitus on helpottaa ja nopeuttaa ohjelmointityötä usealle käyttöjärjestelmälle, jolloin se parantaa myös ohjelmiston siirrettävyyttä huomattavasti.

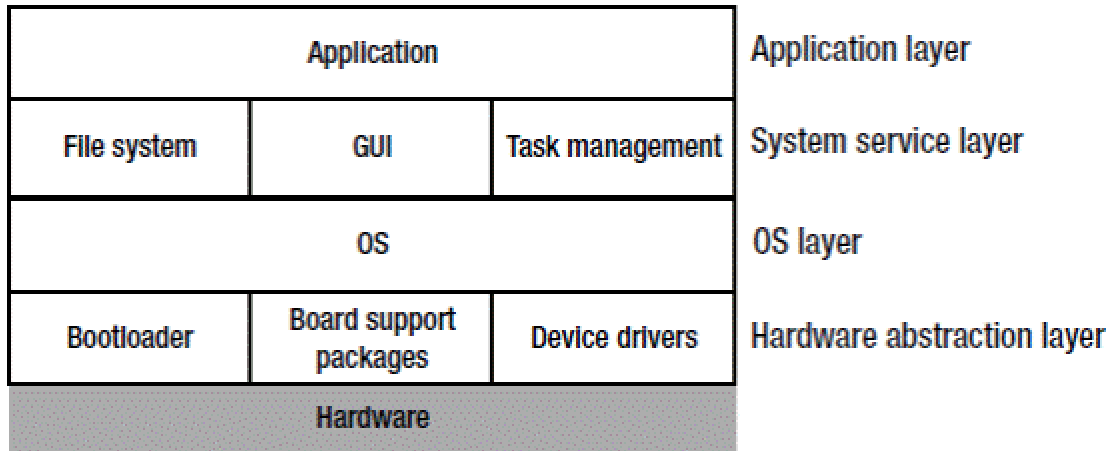
Kokonainen järjestelmä voi sisältää useita rajapintoja, joilla pyritään erottamaan tiettyjä toiminnallisuuksia toisistaan. Järjestelmän abstraktio on esitelty kuvassa 12.



Kuva 12. Järjestelmä abstraktiona [15]

Kuvasta käy ilmi, kuinka laitteisto ja ohjelmisto toimivat omina abstraktiotasoinaan ja kummatkin sisältävät omat sisäiset abstraktionsa. HAL:n tarkoitus on erottaa nämä rajapinnat toisistaan ja ihan ylimmällä sovellustasolla tarkoitus olisi erottaa sovellusabstraktio muusta OSAL-rajapintaa käyttäen.

Myös HAL-rajapinta voidaan jakaa useampaan osaan, jotka näkyvät enemmän laitteiston sisällä. Laitteiston toimintaan vaaditaan erillisiä ajureita, joita HAL-rajapinta pitää sisällään. Ajureita tarvitaan oheislaitteiden käyttämiseen, joten ne tulevat yleensä ohjelmointivaiheessa kehittäjän suunniteltavaksi. Kuvassa 13 on esimerkki, jossa HAL-kerros on jaettu kolmeen osaan.



Kuva 13. Laitteistorajapinnan muodostuminen [16]

Kuvasta käy ilmi HAL-kerroksen sisältö, joka sisältää laiteajureiden lisäksi tärkeän elementin eli alkulatausohjelman (engl. bootloader). Ilman alkulatausohjelmaa laitteisto ei toimi oikein, joten sen tarkoitus onkin laitteiston käynnistyttyä toimia ohjeistuksena laitteistolle, kuinka käynnistytään.

Laiteajureiden on tarkoitus toimia ohjelmistossa omina moduuleinaan, jotka sisältävät toiminnallisuuden erillisten laitteistojen käyttämiseksi. Laiteajureiden on tarkoitus toimia välikätenä niin, että laitteiston ei tarvitse tietää, miten kutakin laitetta ohjataan, vaan jokainen erillinen laiteajuri hoitaa sen. Esimerkiksi mikroaaltouuni, jossa erilliset laiteajurit ohjaavat näppäimistöä, näyttöä, lämpötila-anturia ja säteilyn säätöä. [14.]

4.3.5 Ohjelmiston siirrettävyys

Ohjelmiston siirrettävyyttä tarvitaan tilanteissa, joissa kohdelaitteella on useita toimintaympäristöjä. Mitä enemmän erilaisia laitteita valmistetaan ja mitä enemmän niillä on käyttöympäristöjä, sitä enemmän ohjelmoijan on panostettava siirrettävyyteen. Siirrettävyyteen voidaan panostaa myös siksi, ettei haluta jäädä yhden ainoan valmistajan loukkuun, vaan mahdollisiin uusiin laite- ja ohjelmistoympäristöihin siirtyminen olisi helppoa. Parhaassa tilanteessa ohjelmisto tulisi kirjoittaa siten, että se voidaan siirtää ympäristöstä toiseen ilman muutoksia. Todellisuudessa tällainen on haastava tehtävä, eikä näin voida läheskään aina tehdä. [13, s. 166.]

Ohjelmistojen siirrettävyyteen on useita mahdollisuuksia, ja lopputulos riippuukin aina kokonaisuudesta ja siitä, millaisissa ympäristöissä laitteita tullaan käyttämään. Esimerkkinä muutamia mahdollisia toteutustapoja, jotka ainakin helpottavat siirrettävyyttä.

Ehdollinen käännös

Tässä tarkoituksena on se, että käännösaikaisten symbolien avulla hallitaan käännösprosessia. Symboleilla voidaan esimerkiksi kertoa, mikä käyttöjärjestelmä kohteessa on käytössä, jolloin voidaan generoida tälle käyttöjärjestelmälle sopivaa ohjelmakoodia. Nämä symbolit voidaan määrittää vaikka käyttöjärjestelmäkohtaisissa otsikkotiedostoissa, tai sitten ne voidaan kertoa kääntäjälle komentorivin kautta tai automaattisen käännöstiedoston (esimerkiksi makefilen) kautta. Ehdollista käännöstä sanotaan toimivaksi tavaksi, mutta se tekee koodista helposti sotkuista luettavaa, jonka takia sitä ei pidetä kovin suuressa suosiossa. [13, s. 166.]

Makrot

Makro on kuin pieni aliohjelma, jolloin aliohjelmaa kutsuttaessa kääntäjä generoikin makron tämän kyseisien ohjelmakutsun tilalle. Esimerkiksi C-kielen printf-funktio on todellisuudessa makro. Tämä makro onkin varmasti tuttu kaikille kielellä ohjelmoiville. [13, s. 167.]

5 IO-kortin esittely

Isona osana tätä työtä on mikrokontrolleri, mille ohjelmistoja tullaan kehittämään. Loppujen lopuksi mikrokontrolleri tulee määräämään suurimman osan siitä, kuinka kehitysympäristön komponentit tulisi valita. Mikrokontrollereille ohjelmistojen kehittäminen on hyvin laitekohtaista, joten valittavat ratkaisut pohjautuvat pitkälti aina mikrokontrollerin ominaisuuksiin. Tässä työssä käytettävä mikrokontrolleri on ARM-pohjainen STM32.

Mikrokontrolleri STM32 on STMicroelectronicsin valmistama. STMicroelectronics on yksi maailman suurimpia puolijohteita valmistavia yrityksiä [17]. STM32 on 32-bittinen mikrokontrolleriperhe, joka perustuu ARM-arkkitehtuuriin ja Cortex-M-luokan prosessoriin. STM32-tuoteperheeseen kuuluu useita erilaisia toteutuksia, ja ne jakautuvat neljään pääryhmään: suorituskykyisiin, kustannustehokkaisiin, langattomiin ja alhaisen virrankulutuksen omaaviin [18]. Tässä työssä hyödynnetään vähävirtaisiin piireihin kuuluvaa STM32L476RG:tä. Mikrokontrollereiden ja prosessoreiden nimet ja lyhenteet voivat olla hämmentäviä, joten esitellään lyhyesti, mitä nämä tarkoittavat:

- 32-bittinen, kun puhutaan johonkin elektroniikkaan liittyvän laitteen bittisyydestä, sillä viitataan yleensä suorittimeen. Suorittimen bittisyydellä tarkoitetaan muistipaikkojen, rekisterien ja väylien leveyttä, joka taas vaikuttaa johdannaisesti suorittimen suorituskykyyn. Esimerkiksi STM32 on 32-bittinen, jolloin se pystyy muistin ja suorittimen välillä siirtämään 32 bittiä leveän sarjan kerrallaan.

- ARM-arkkitehtuuri on 32-bittinen mikroprosessoriarkkitehtuuri, jonka on kehittänyt brittiläinen tietokonevalmistaja Acorn. Yleisesti termillä ARM tarkoitetaan nykypäivänä ARM-arkkitehtuurin omaavia mikrokontrollereita ja prosessoreita, joita useat eri valmistajat tuottavat. ARM on siis arkkitehtuuri, jota kuka tahansa valmistaja voi käyttää omassa prosessoritoteutuksessaan, kuten tässä tapauksessa STMicroelectronics hyödyntää sitä omassa STM32-tuoteperheessään. ARM ei siis varsinaisesti valmista prosessoreita, vaan myy lisensseinä omaa arkkitehtuuriaan. Esimerkiksi yritys, joka valmistaa mikrokontrollereita, haluaa käyttää Cortex-M-prosessoriarkkitehtuuria. Se joutuu ostamaan lisenssin saadakseen käyttää tätä arkkitehtuuria, jolloin ARM toimittaa kyseisen prosessoriarkkitehtuurin lähdekoodin Verilog-HDL kielellä (engl. Hardware Description Language) yritykselle. Tämän jälkeen yritys lisää mikrokontrolleriarkkitehtuuriin omat lisäyksensä, esimerkiksi muistia ja AD-muuntimia [10, s. 6].

- Cortex-M-sarjan prosessorit on pääasiassa suunniteltu sulautettuihin laitteisiin, koska ne ovat erittäin halpoja ja energiatehokkaita. Cortex-sarjaan kuuluu muitakin prosessoreita, kuten esimerkiksi useassa älypuhelimessa hyödynnettävää Cortex-A-sarjaa.

5.1 ARM-mikrokontrollerin ominaisuuksia

ARM-arkkitehtuuri on kehitetty Reduced Instruction Set Computer (RISC)-suoritinarkkitehtuurifilosofiaa käyttäen, eli sen konekieliset käskyt on pyritty pitämään mahdollisimman yksinkertaisina. Arkkitehtuurista käytetään myös nimitystä load-and-store, joka tarkoittaa datan siirtämistä muistien ja suorittimen rekisterien välillä [21, s.4].

Se, minkälaista toteutusta missäkin sulautetussa laitteessa tullaan käyttämään, on aina hyvin tapauskohtaista, mutta mikrokontrollerin käyttämisessä ja etenkin ARM-arkkitehtuuriin omaavien Cortex -M-mikroprosessoreiden käyttäminen antaa tiettyjä hyötyjä. Alla esitetään muutamia hyötyjä, joita Cortex-M-mikrokontrollerilla on tarjota.

Vähäinen virrankulutus

Moneen muuhun 32-bittiseen mikrokontrolleriin verrattuna Cortex-M-mikroprosessorit kuluttavat huomattavan vähän virtaa. Prosessorit on optimoitu kuluttamaan vähän virtaa, ja niillä voidaan päästä jopa alle 100 µA/MHz virran kulutukseen. Tämä antaa mahdollisuuden käyttää tätä arkkitehtuuria myös vähävirtaisten sovellusten kanssa. [10, s. 8.]

Suorituskyky

Näissä prosessoreissa huomattavaa on myös niiden suorituskyky, joka mahdollistaa niiden käyttämisen useissa tehoa vaativissa sovelluksissa. Ja taas vastaavasti kellotaajuutta voidaan hidastaa tarpeen mukaan, jolloin myös virrankulutus pienenee. [10, s. 9.]

Energiatehokkuus

Kaksi ylempää ominaisuutta yhdistettynä voidaan puhua erittäin energiatehokkaasta paketista. Eli tarpeen vaatiessa voidaan suorittimelta ottaa paljon laskentatehoa, kun sitä tarvitaan. Vastaavasti, kun sitä ei tarvita, voidaan prosessori asettaa sleep-moodiin, jolloin virrankulutus on mitätöntä.

Ohjelmiston suuruus

ARM-prosessorien käyttämä Thumb-käskykanta mahdollistaa sen, että ohjelmistojen koot pysyvät pieninä, mikä taas vaikuttaa virrankulutukseen parantavasti ja myös siihen, ettei muistia tarvitse käyttää paljon. [10, s. 9.]

Skaalautuvuus

Cortex-M-prosessorit skaalautuvat helposti, koska niistä on useita erilaisia toteutuksia. Vaihtoehtoja on todella edullisista mikrokontrollereista todella tehokkaisiin, jolloin toteutukset erilaisissa ympäristöissä helpottuvat. Tällaisen prosessoriarkkitehtuurin johdonmukaisuuden ansiosta ohjelmistojen kehittämiseen ei tarvita useita erilaisia työkaluja, jolloin se antaa mahdollisuuden kierrättää vanhoja ohjelmistoja uudelleen.

Ohjelmiston siirrettävyys ja uudelleenkäyttö

ARM-arkkitehtuuri on erittäin C-kieliystävällistä, mikä mahdollistaa sen, että kaiken voi ohjelmoida ANSI C-kieltä käyttäen. Myös ARM:n tarjoama CMSIS helpottaa Cortex-M-prosessorien ohjelmointia tarjoamalla API:n, jolla päästään käsiksi prosessorin ominaisuuksiin. Tämä lisää ohjelmiston uudelleenkäytettävyyttä ja ohjelmistojen siirrettävyyttä eri sovelluksien välillä. [10, s. 10.]

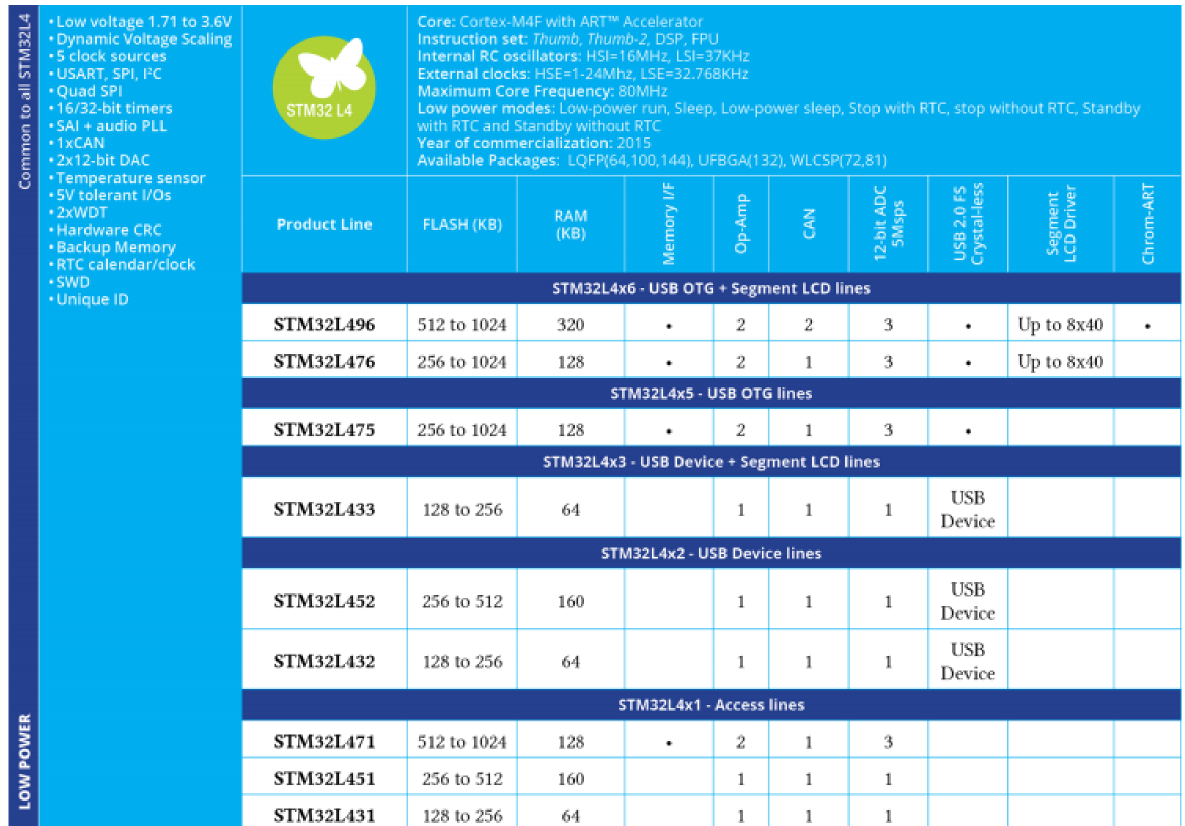
Useita vaihtoehtoja

Cortex-M-mikrokontrollereita on useilla eri valmistajilla omilla toteutuksillaan, jolloin valinnan varaa on useita. Myös ohjelmistonkehitykseen olevia työkaluja on useita erilaisia, kuten virheenetsintätyökalut, ohjelmointiympäristöt, väliohjelmistot ja sulautetut käyttöliittymät. [10, s. 10.]

5.2 Mikrokontrolleri

Tähän työhön STMicroelectronics on valittu ensisijaiseksi toimittajaksi, koska tuki tuotekehitykseen on erinomainen. STMicroelectronics tukee hyvin esimerkiksi useita ohjelmointiympäristöjä, STlink- ja JTAG-debuggausta raudassa ja esimerkkikoodien ja valmiiden vapaasti käytettävien kirjastojen saatavuus on hyvä.

Työssä käytettävä mikrokontrolleri on alhaisen virrankulutuksen omaava STM32L476RG. Kuvassa 14 esitellään edellä mainitun piirin ja samaan sarjaan kuuluvien piirien ominaisuuksia.



Common to all STM32L4

- Low voltage 1.71 to 3.6V
- Dynamic Voltage Scaling
- 5 clock sources
- USART, SPI, I²C
- Quad SPI
- 16/32-bit timers
- SAI + audio PLL
- 1xCAN
- 2x12-bit DAC
- Temperature sensor
- 5V tolerant I/Os
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

LOW POWER

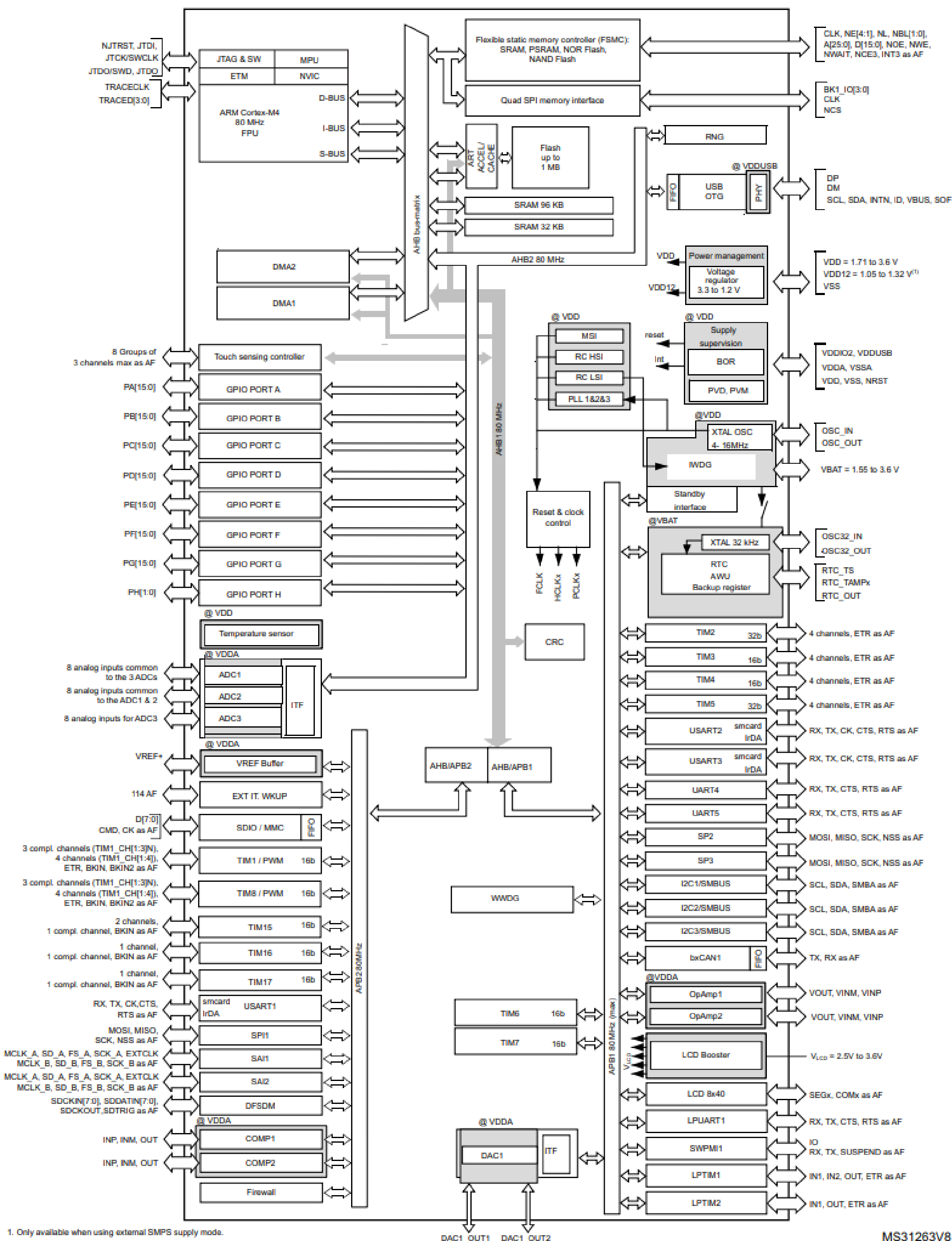
STM32 L4

Core: Cortex-M4F with ART™ Accelerator
Instruction set: *Thumb*, *Thumb-2*, DSP, FPU
Internal RC oscillators: HSI=16MHz, LSI=37KHz
External clocks: HSE=1-24Mhz, LSE=32.768KHz
Maximum Core Frequency: 80MHz
Low power modes: Low-power run, Sleep, Low-power sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC
Year of commercialization: 2015
Available Packages: LQFP(64,100,144), UFBGA(132), WLCSP(72,81)

Product Line	FLASH (KB)	RAM (KB)	Memory I/F	Op-Amp	CAN	12-bit ADC 5Msps	USB 2.0 FS Crystal-less	Segment LCD Driver	Chrom-ART
STM32L4x6 - USB OTG + Segment LCD lines									
STM32L496	512 to 1024	320	•	2	2	3	•	Up to 8x40	•
STM32L476	256 to 1024	128	•	2	1	3	•	Up to 8x40	
STM32L4x5 - USB OTG lines									
STM32L475	256 to 1024	128	•	2	1	3	•		
STM32L4x3 - USB Device + Segment LCD lines									
STM32L433	128 to 256	64		1	1	1	USB Device		
STM32L4x2 - USB Device lines									
STM32L452	256 to 512	160		1	1	1	USB Device		
STM32L432	128 to 256	64		1	1	1	USB Device		
STM32L4x1 - Access lines									
STM32L471	512 to 1024	128	•	2	1	3			
STM32L451	256 to 512	160		1	1	1			
STM32L431	128 to 256	64		1	1	1			

Kuva 14. L4-tuoteperheen ominaisuuksia [19]

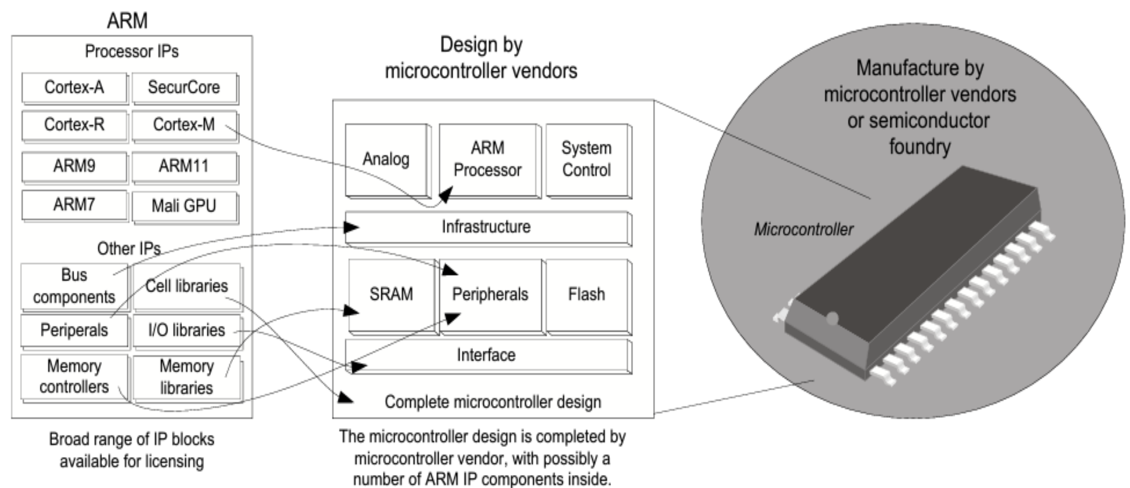
L476RG:ssä on alhainen virrankulutus, ja siinä on jopa 80 MHz kellotaajuudella toimiva Cortex-M4-prosessori. Cortex-M4-prosessorit on suunniteltu ARMv7-M-arkkitehtuurin mukaisesti. Kuvassa 15 on esitelty STM32L476-piirin lohkokaavio.



Kuva 15. STM32L476-lohkokaavio [20]

Lohkokaaviosta nähdään piirin sisältämät toiminnot ja myös piirin periaatteellinen toiminta. Ominaisuuksissa huomioitavaa on, että useat ominaisuudet tulevat prosessorilta, eli tässä tapauksessa Cortex-M4:ltä. Kyseistä prosessoria käyttävän valmistajan mikrokontrolleri sisältää siis Cor-

tex-M4:n ominaisuuksien lisäksi omat valmistajakohtaiset ominaisuutensa, kuten tässä tapauksessa STMicroelectronicsin. Tällöin esimerkiksi muistikartat, kellotaajuudet ja jännitteet voivat poiketa toisistaan. Kuvassa 16 on esitelty tämä periaate. Kuvassa mikrokontrollerin toteutukseen valitaan ARM-prosessorin lisäksi muitakin ARM:n lisensoimia tuotteita, joita ARM:lla on tarjottavana.



Kuva 16. Tuotelisensseistä mikrokontrolleriksi [10]

5.2.1 Mikrokontrollerin rajapinta

STMicroelectronicsilla on STM32-mikrokontrollereille oma rajapintansa, jonka kautta päästään helposti käsiksi mikrokontrollerin ominaisuuksiin. Rajapinta on jaettu kahteen tasoon, ylempään Hardware Abstraction Layer (HAL)-tasoon ja alempaan Low-Layer (LL)-tasoon. STMicroelectronics itse suosittelee käyttämään rajapintaa oman STM32Cube-sovelluksensa kautta. Sovelluksen on tarkoitus helpottaa kehittämistyötä sen tarjoaman graafisen käyttöliittymän avulla, jonka kautta kehittäjä pystyy helposti generoimaan valmista koodia. [21.]

HAL:n tarkoitus on toimia ohjelmointirajapintana mikrokontrollerin ja kehitysympäristön välillä. HAL toimii niin sanottuna ylempänä kerroksena. HAL-rajapinta voidaan jakaa kahteen osaan, geneeriseen ja laajennusosioon. Geneerisen osion tarkoitus on tarjota yleiset ja geneeriset funktiot kaikille STM32-sarjan mikrokontrollereille. Laajennusosan tarkoitus on olla yksityiskohtaisempi rajapinta tietylle mikrokontrollerille, jolloin se tarjoaa kustomoituja funktioita, jotka toimivat ainoastaan tietylle piirille. [21.]

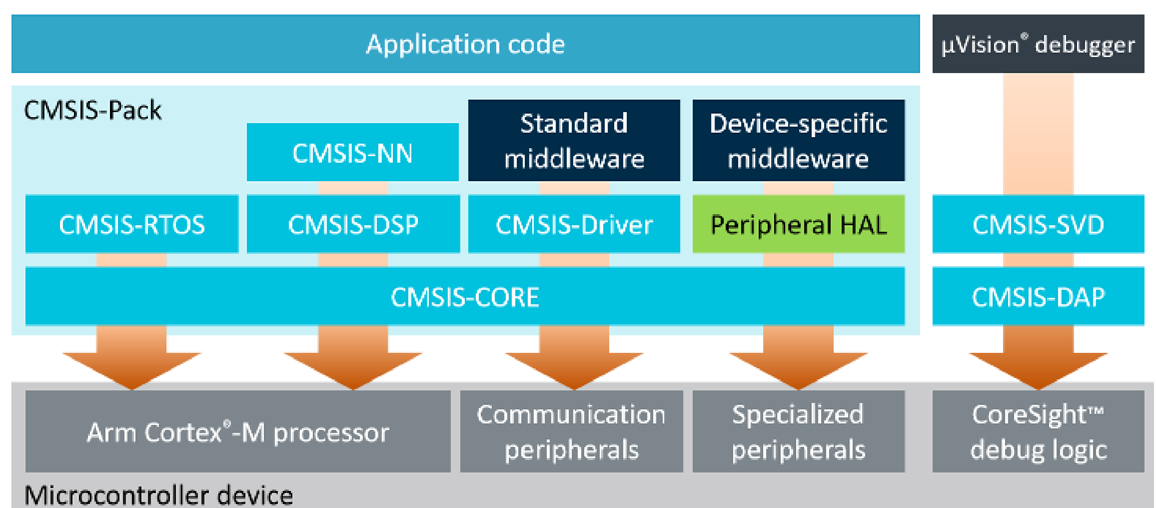
Low-layer toimii nimensä mukaisesti ohjelmointirajapintana enemmän atomisella tasolla, eli tällöin ohjelmoidaan niin sanotusti lähempänä rautaa. LL:n tarkoitus on tarjota nopea ja kevyt ohjelmistonkehityskerros, mutta sen käyttäminen vaatii enemmän perehtyneisyyttä verrattuna HAL-kerrokseen. [21.]

Käytön kannalta HAL tarjoaa korkean tason ohjelmointirajapinnan, joka takaa helpomman siirrettävyyden ohjelmistolle, HAL myös pyrkii piilottamaan käyttäjältä mikrokontrollerin ja lisälaitteiden käytön hankaluuden. LL taas tarjoaa alemman tason ohjelmointirajapinnan suoraan rekisteritasolla, jolla voidaan paremmin optimoida koodia, mutta tällöin ohjelmiston siirrettävyys kärsii huomattavasti. LL-käyttö vaatii myös huomattavasti enemmän perehtyneisyyttä, koska ollaan tekemisissä rekistereiden kanssa. [21.]

HAL- ja LL-rajapinnat on molemmat kirjoitettu käyttäen tiukennettua C-kieltä. Tällöin noudatetaan siis C-kielen standardia erittäin tiukasti, jolloin mahdolliset ongelmat eri C-kieliversioiden välillä jäisivät mahdollisimman vähäisiksi. [21.]

5.2.2 ARM-rajapinta

ARM-arkkitehtuuri mahdollistaa CMSIS:n käyttämisen, joka on ARM:in kehittämä ohjelmistoinfrastruktuuri. ARM:n tarkoitus CMSIS:n kehittämisessä on yrittää saada standardi ohjelmistonkehityksen ja mikrokontrollereiden valmistajien välille, jolloin ohjelmistojen siirrettävyys ja uudelleenkäyttö olisi helpompaa. Kuvassa 17 on esitelty CMSIS-rakenne.



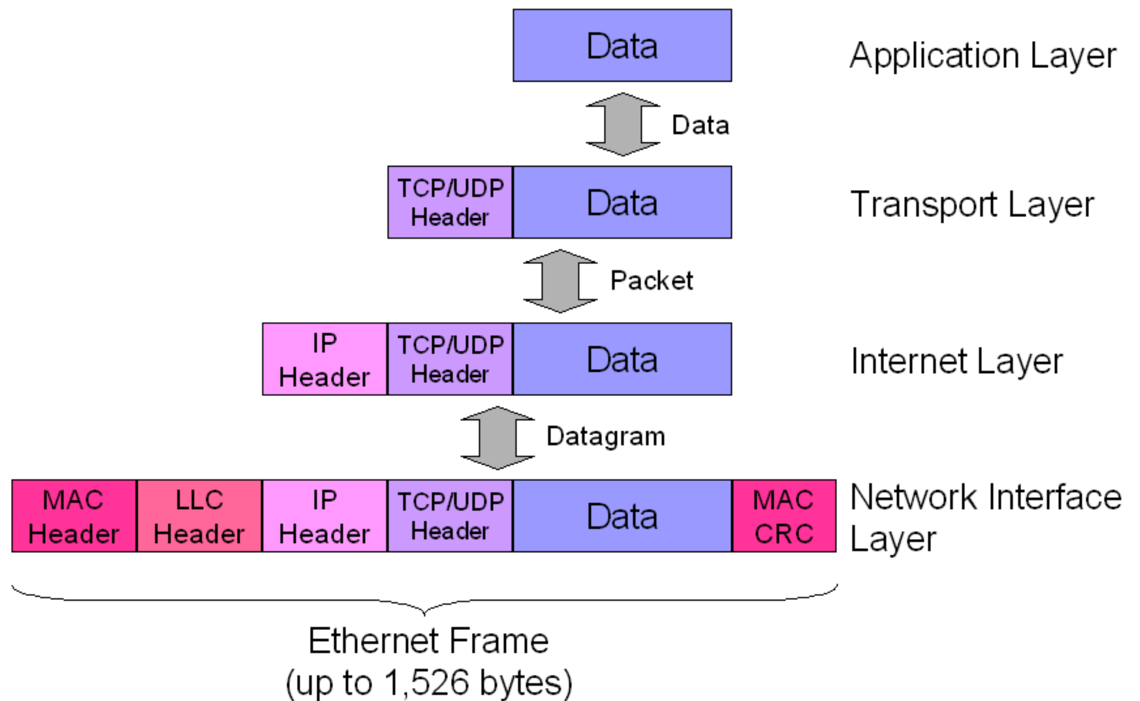
Kuva 17. CMSIS [23]

Kuten kuvasta käy ilmi, CMSIS koostuu useammasta eri komponentista. Jokaisella komponentilla on omat tehtävänsä, mutta Cortex-M-mikrokontrollereille tärkein komponentti on CMSIS-CORE. CMSIS-COREn tarkoituksena on antaa sovelluskehittäjille ohjelmointirajapinta Cortex-M-mikroprosessoreille riippumatta siitä, kenen valmistajan mikrokontrolleri taustalla on. [10, s. 48.]

5.3 Tietoliikenne

IO-kortit sisältävät myös tietoliikenneosion, jonka tarkoitus on toimia tietoväylänä IO-kortin ja päätietokoneen välillä. Tietoliikenteenä käytetään pakettipohjaiseen lähiverkkoratkaisuun perustuvaa Ethernet-lähiverkkotekniikkaa. Ethernetia käytetään teollisuusympäristöissä sen luotettavuuden takia.

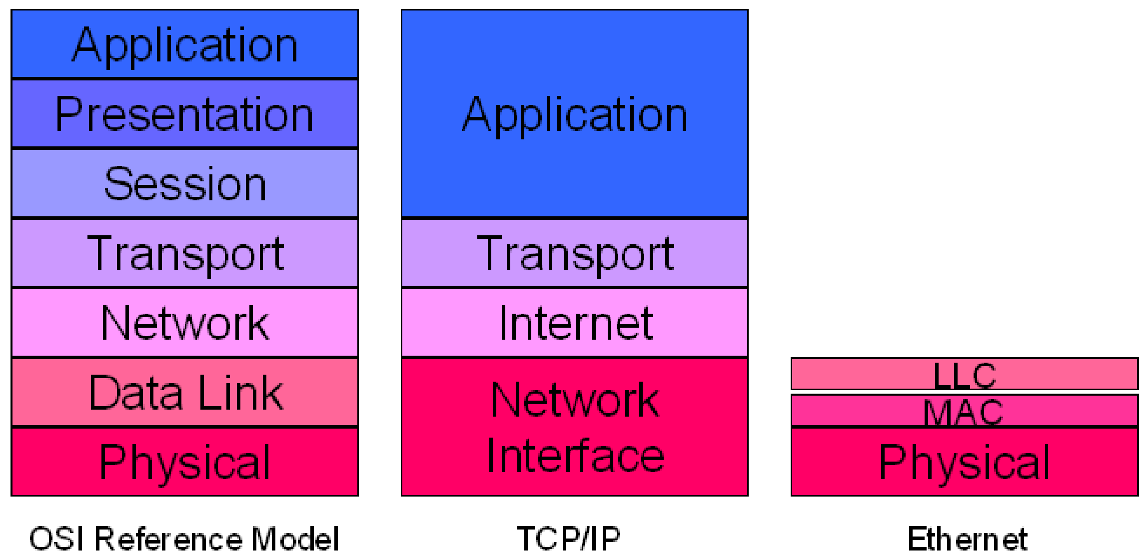
Toimintaperiaatteeltaan Ethernet perustuu kehyksiin, joissa laitteet keskustelevat keskenään lähettämällä ja vastaanottamalla kehyksiä muilta samassa verkossa olevilta laitteilta. Kehys sisältää lähetettävän datan lisäksi osoitetietoja, kuten MAC-osoitteita, joista käy ilmi, mikä laite lähettää viestiä ja mille laitteelle viesti on tarkoitettu. Kehys sisältää myös tietotyypin siitä, mitä protokollaa datan liikuttamiseen käytetään, tai jos erillistä protokollaa ei ole määritelty, se sisältää tiedon kehyksen koosta. Kehyksen loppuun lisätään tarkistussumma, jonka tarkoituksena on antaa tieto vastaanottopäähän, onko tiedon lähettämisen aikana data jollain tavalla korruptoitunut. [24.] Kuvassa 18 on esitelty, kuinka kehys muodostuu lähetettäväksi paketiksi.



Kuva 18. Ethernet-kehys [24]

Kuvasta käy ilmi, mistä kehys muodostuu ja kuinka sen liikkuminen kerrosten välillä tapahtuu TCP/IP-protokollapinoa käyttäen. Lopussa kehyksestä muodostuu lähetettävä paketti, jonka siirto tapahtuu IP-osoitteen perusteella. Tiedon kuljettamiseen voidaan käyttää eri tietoliikenneprotokollavaihtoehtoja. Kuvan esimerkissä tiedon kuljettaminen tapahtuu TCP- tai UDP-protokollan avulla. Näiden kahden periaatteellinen ero on siinä, että TCP:n on tarkoitus pitää huolta siitä, että paketit saapuvat perille oikeassa järjestyksessä ja niin, että kaikki paketit pääsevät perille. UDP-protokollassa paketit vain lähetetään kohdeosoitteeseen välittämättä siitä, onko vastaanottopäässä ketään, eikä varmuutta ole siitä, ovatko paketit saapuneet perille ollenkaan tai ovatko ne korruptoituneet matkalla.

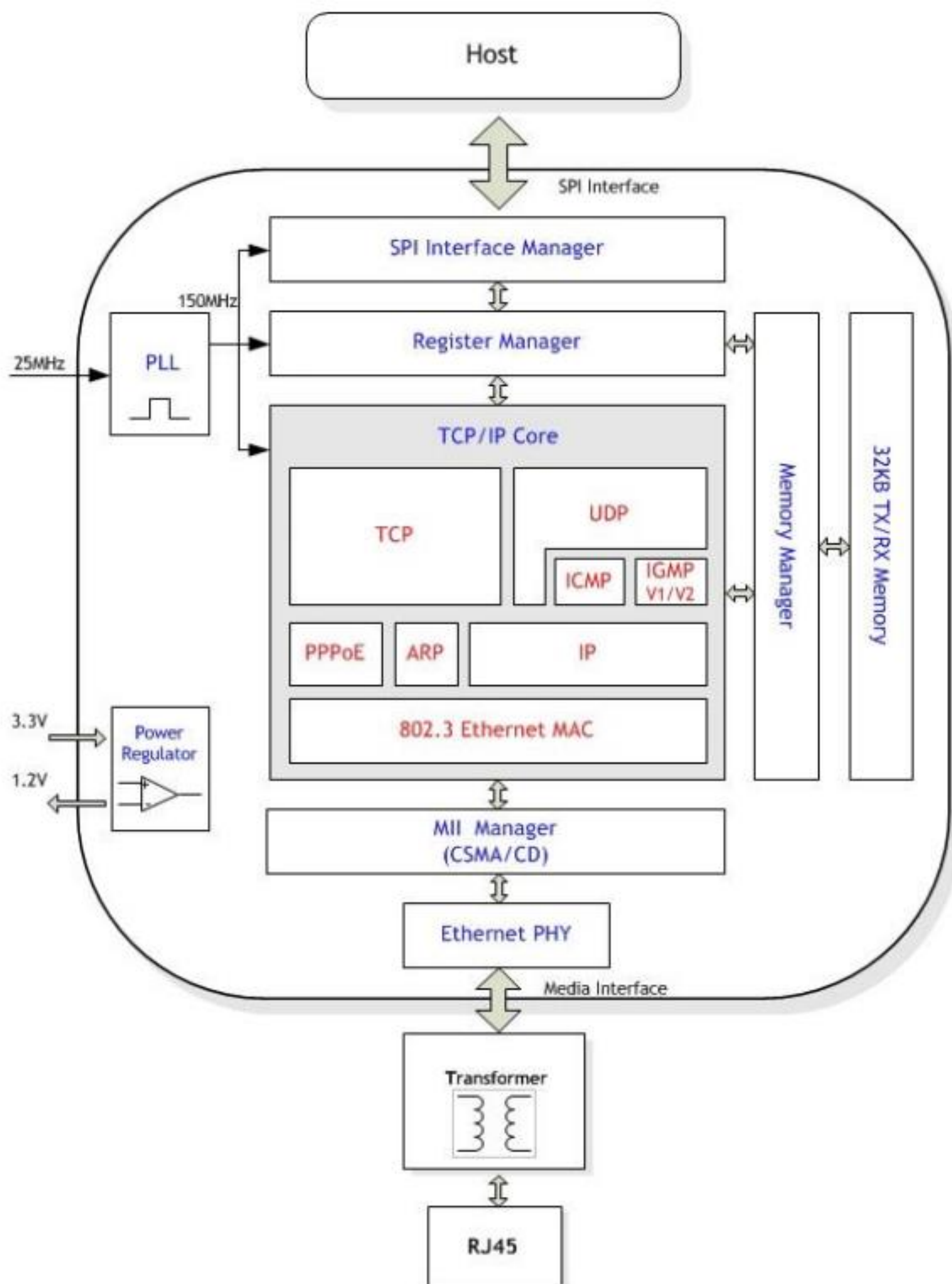
Tietoliikenteen toiminta on monivaiheinen prosessi, jossa tieto liikkuu useamman kerroksen välillä. Tiedon liikkuminen eri kerrosten välillä riippuu aina toteutuksesta, mutta tässä työssä data liikkuu Ethernetin ja TCP/IP-protokollapinon avulla. Kuvassa 19 on esitelty Ethernet-arkkitehtuuri.



Kuva 19. Ethernet-arkkitehtuuri [24]

Kuvasta käy ilmi Ethernetin muodostamat kerrokset: LLC-, MAC- ja fyysinen kerros. LLC-kerros eli Logic Link Control-kerros on vastuussa siitä, että lähtevät kehykset sisältävä tiedon lähetykseen käytettävästä protokollasta. MAC-kerros eli Media Access Control on vastuussa lähetettävän kehyksen kasaamisesta. Tämä kerros vastaa myös siitä, että kehys sisältää vastaanottavan koneen ja lähettävän koneen MAC-osoitteet. Fyysisen kerroksen tarkoituksena on muuntaa MAC-kerroksen tuottama kehys sähköksi tai sähkömagneettisiksi aalloiksi, riippuen siitä, käytetäänkö langatonta vai langallista tiedonsiirtoa. [24.]

Tässä työssä käytettävä mikrokontrolleri ei itsessään sisällä mahdollisuutta implementoida TCP/IP-protokollapinoa, joten sitä varten käytetään erillistä Ethernet-ohjainta. Ethernet-ohjaimena käytetään WIZnet-nimisen yrityksen valmistamaa W5500-piiriä, joka tarjoaa valmiin TCP/IP-protokollapinon. Piiri keskustelee mikrokontrollerin kanssa SPI-väylää käyttäen, jota kautta piiri myös ohjelmoidaan. Kuvassa 20 on esitelty W5500-piirin lohkokaavio.



Kuva 20. W5500-lohkokaavio [25]

Lohkokaaviosta käy ilmi TCP/IP-protokollapinon muodostama alue ja siihen liittyvät muut komponentit, kuten muisti, SPI-rajapinta ja Ethernetin muodostama fyysinen rajapinta, joka sisältää myös RJ45-liitännän tiedonsiirtoa varten.

6 Kehitysympäristön toteutus

Kehitysympäristön toteutus tehdään tutkimus- ja kehitysmenetelmiä käyttäen, ja se jaetaan neljään eri vaiheeseen: vaatimusmäärittelyyn, suunnitteluun, toteutukseen ja testaukseen. Nämä vaiheet käsitellään seuraavissa aliluvuissa.

6.1 Vaatimusmäärittely

Kehitettävän kehitysympäristön tulisi sisältää kaikki tarpeellinen ohjelmistonkehitystyötä ajatellen. Virtuaalikoneeseen rakennetaan kehitysympäristö, joka on kopioitavissa kehittäjältä toiselle. Ympäristö myös dokumentoidaan niin, että kehittämisen aloittaminen uudella ympäristöllä on helppoa ja nopeaa. Myös käytössä oleva tietoliikennekirjasto täytyy muokata toimivaksi uuteen ympäristöön.

Kehitysympäristöä suunniteltaessa tulee huomioida laitteisto, jolle ohjelmistoja tullaan kehittämään. Laitteistoksi on kaavailtu STM32-mikrokontrolleria, joka sisältää ARM cortex-M4-prosessoriarkkitehtuurin.

Mikrokontrollerin tulisi myös pyörittää nykyistä tietoliikennettä mahdollisimman pienin muutoksin. Tämä tulisi muokata ja testata kehitysympäristössä. Ohjelmistonkehitystyötä ajatellen ohjelmointikielenä voidaan käyttää ANSI C:tä ja tarpeen vaatiessa C++:aa, jolloin mahdolliset rajapintojen väliset erot jäävät mahdollisimman pieniksi.

6.2 Suunnittelu

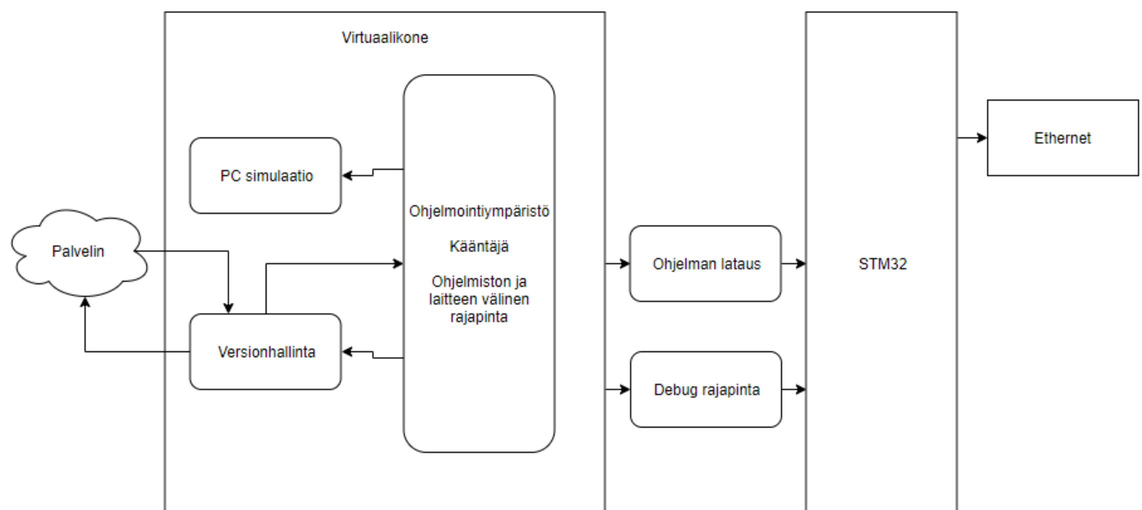
Vaatimusmäärittelyn perusteella voidaan aloittaa kehitysympäristön suunnittelu. Koska kehitysympäristö suunnitellaan uudelle mikrokontrolleriperheelle, on suunnittelu helpointa aloittaa rakenteellisella suunnittelulla ja tutkimalla eri komponenttivaihtoehtoja, jolloin heti suunnittelun alussa saadaan rajattua kehitysympäristöön liittyviä osia.

Kehitysympäristöä suunniteltaessa tulee huomioida useita eri tekijöitä. Suurin määräävä tekijä on mikrokontrolleri, jolle kehitystyötä tullaan tekemään. Toinen suuri tekijä on se, millä mikrokontrolleri tullaan ohjelmoimaan, koska ohjelmointiympäristöllä täytyy olla tuki tälle rajapinnalle.

Karkeasti listattuna kehitysympäristön komponentit:

- Virtuaalikone, Linux tai Windows
- Kääntäjä
- Laitteiston ja ohjelmiston välinen rajapinta
- Ohjelmointiympäristö ohjelmistonkehitystä ja projektin hallintaa varten
- Ohjelman lataus mikrokontrollerille
- Virheen etsintä rajapinta (engl. Debug)
- Versionhallintaan koodit ylläpitoa varten
- Ohjelmointiympäristö PC-simulaatiota varten

Kuvassa 21 on esitelty kehitysympäristön suunniteltu rakenne.



Kuva 21. Kehitysympäristön rakenne

Kehitysympäristö

Vaatimusmäärittelyn mukaan kehitysympäristö suunnitellaan virtuaalikoneeseen, joka on Windows tai Linux riippuen kehitysympäristön komponenttien toimivuudesta kyseisissä ympäristöissä.

Versionhallinta

Kajaanissa on käytössä Apache subversion-versionhallintajärjestelmä, joten sen hyödyntämiseen on asennettava asiakasohjelma virtuaalikoneelle, joka on yhteydessä versionhallintaa pyörittävään palvelimeen.

PC-simulaatio

PC-simulaation toteuttamiseen voi olla paljon valmiita ratkaisuja, mutta myös omien simulaattoreiden suunnittelu on mahdollista, jolloin siihen tarvitaan omat työkalunsa.

Tietoliikenne

Tietoliikenteenä työssä käytetään Ethernetia ja vaatimusmäärittelyn mukaan nykyinen käytössä oleva tietoliikennerajapinta täytyy implementoida uuteen ympäristöön. Tässä on huomioitava myös käyttöön tuleva Ethernet-piiri, joka voi vaikuttaa implementointiin. Rajapinnan implementoinnissa täytyy huomioida myös, että vaatimusmäärittelyssä määriteltiin, ettei käytössä olevaa kirjastoa tulisi muokata liikaa.

Ohjelmointiympäristö

Ohjelmointiympäristö eli niin sanotun IDE:n toteutus voi olla kokonaan valmispaketti, jolloin ympäristö sisältää itsessään kääntäjän, koodinlatauksen mikrokontrollerille ja jopa simulointimahdollisuuden. Toinen vaihtoehto on, että nämä palaset täytyy kasata itse. Ohjelmointiympäristömahdollisuuksia on useita, joista toiset ovat avoimen lähdekoodiin perustuvia eli ilmaisia ympäristöjä, kun taas toiset maksullisia. Tällaisten asioiden vuoksi ympäristön valintaa tehdessä jokaista ympäristöä ei voida testata, vaan testattavat ympäristöt täytyy valita aluksi hyvin perusteluin.

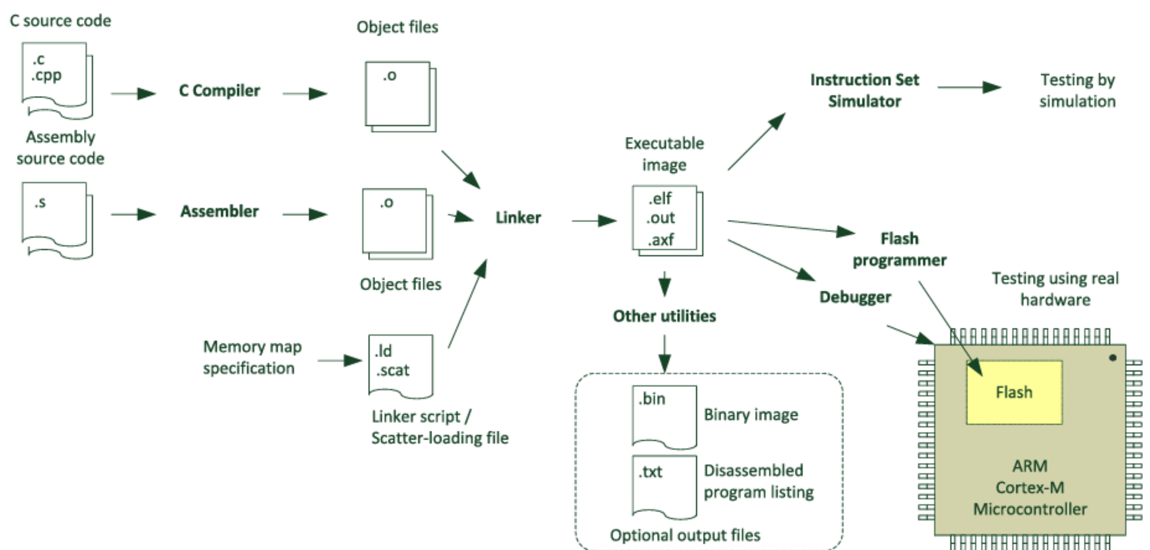
6.3 Kehitys

Suunnittelun jälkeen voidaan aloittaa kehitysympäristön toteuttaminen. Kehityksen tavoitteena on luoda testattavien kehitysympäristöjen rakenne valmiiksi suunnittelu- ja tutkimustyön mukaisesti. Suunnittelun ja tutkimisen perusteella saadun tiedon avulla voidaan valita testattavat ohjelmointiympäristöt ja muut komponentit.

Kehitystyötä tehdessä tehdään myös pieniä testauksia, joissa käytetään apuvälineenä STMicroelectronicsin Nucleo-kehitysalustaa. Tässä kehitysalustassa on mikroprosessorina aikaisemminkin mainittu STM32L476RG. Testauksessa hyödynnetään myös Ethernet-moduulia, joka sisältää W5500-piirin. Moduuli voidaan kytkeä kehitysalustaan helpoiten kytkentäalustaa ja Dupont-johtoja apuna käyttäen.

6.3.1 Ohjelmointiympäristö

Kehitysympäristö sisältää siis useita komponentteja, joista suurin osa on nykyään upotettu ohjelmointiympäristöön (engl. Integrated development environment). Esimerkiksi kääntäjä, ohjelman lataustyökalut mikrokontrollerille, virheenetsintärajapinta (engl. debug) ja joissakin tapauksissa myös versionhallinta. Ohjelmointiympäristön toiminnan periaate on esitelty kuvassa 22.



Kuva 22. Ohjelmointiympäristön toiminnan periaate [10]

Ohjelmointiympäristössä käyttäjä kirjoittaa C- tai C++-kielellä ohjelmaa eli lähdekoodia. Seuraavaksi kääntäjä tekee tiedostoista objektitiedostoja, jotka ovat käytännössä konekieltä, mutta niitä ei voi itsessään suoraan ajaa. Myös mahdollisesta konekielisestä koodista assembler luo erikseen omat objektitiedostonsa. Tämän lisäksi tarvitaan linker-skripti, joka sisältää muistikartan ja ohjeistuksen siitä, kuinka objektitiedostot tulee kasata. Ennen valmista ajettavaa tiedostoa linker yhdistää nämä kaikki tiedostot, jolloin saadaan jokin suoritettava tiedosto ja mahdollisesti binaari tai heksadesimaali image, jotka voidaan myös ladata mikrokontrollerille. Esimerkiksi STM32:lle ladattaessa ohjelmia voidaan binaaritiedosto vain tiputtaa Flash-muistille niin kuin normaalille

muistitikulle, jolloin ohjelma lähtee suoriutumaan mikrokontrollerilla. Ohjelman lataus laitteelle voidaan siis suorittaa usealla tavalla, mutta yleisin tapa siihen on käyttää debuggeria (engl debugger) eli samaa laitetta ja rajapintaa kuin virheitä etsiessä (engl. debugging). Tässä vaiheessa voidaan ohjelmaa ajaa myös simulaattorilla, joka osaa simuloida ARM-arkkitehtuurin mukaista Thumb-ohjeistusta.

Cortex-M-tuotteille on useita eri ohjelmointiympäristövaihtoehtoja maksullisista ilmaisiin vaihtoehtoihin. Suurin osa ilmaisista vaihtoehdoista on Eclipse pohjaisia, eli ne on rakennettu jonkin tietyn Eclipse-version päälle. Myös osa maksullisista on suunniteltu Eclipseä käyttäen. Eclipse on avoimen lähdekoodin ohjelmointiympäristö, jota voi käyttää suoraan ohjelmointiympäristönä tai siitä voidaan räätälöidä omanlaisensa paketti tietyn laitteen kehitystä varten.

Kehitystyötä tehdessä testataan lyhyesti useampia eri ohjelmointiympäristöjä. Testaukseen valitaan maksullisia ja ilmaisia ohjelmointiympäristöjä, jotta voidaan valita lopulliseen testaukseen tulevat ympäristöt. Ympäristön testaus on tärkeä tekijä, mutta niiden paljouden vuoksi ei kaikkia voida kuitenkaan testata pitkän kaavan mukaan. Lyhyellä testauksella saa nopeasti yleiskuvan itse ympäristöstä, jonka perusteella voidaan valita haluttavat ominaisuudet. Alla listattuna ohjelmointiympäristöt, joita kehitystä tehdessä kokeiltiin.

- Arduino IDE
- ARM DS-5
- IAR Systems embedded workbench
- mbed.org
- Atollic Truestudio for STM32
- Eclipse
- Visual Studio
- ARM Keil µVision MDK

Lyhyessä testauksessa katsotaan ympäristön yleistä ilmettä, projektinhallintaa, tarvitaanko mikrokontrollerin kehittämiseen jotain lisäosia ja kuinka hyvin ympäristö tukee STM32-mikrokontrollereita ja ARM-prosessori-arkkitehtuuria. Samalla testataan esimerkkikoodeja, jotta nähdään, kuinka ympäristö toimii ja onko koodin lataus ja virheenetsintä tuettu hyvin.

Menemättä sen tarkemmin jokaisen ohjelmointiympäristön testauksen tuloksiin voidaan yleisesti sanoa, että jokaisessa ympäristössä on omat hyvät ja huonot puolensa. Esimerkiksi Arduino IDE

on helppo ja yksinkertainen ja sopii varsin hyvin kevyeen prototyyppien tekemiseen. Arduinosta helppoa tekee se, että se piilottaa taakseen kaiken vaikean säätelyn ja väylien alustamisen, mutta se on projektinhallinnallisesti huono. Toisena esimerkkinä voidaan mainita Eclipse ja Visual studio, jotka toimivat mainiosti mikrokontrollerin kanssa, mutta nämä ympäristöt vaativat useita lisäosia, jotta ohjelmistoja voidaan kehittää STM32 mikrokontrollereille. Lopulliseen testaukseen näistä ympäristöistä valitaan Atollic Truestudio ja Keil µVision. Alla on esitetty tietoja lyhyesti kyseisistä ympäristöistä ja myös perusteluja, miksi valinta kohdistuu kyseisiin ympäristöihin.

Atollic Truestudio for STM32

Kuten ohjelmointiympäristön nimikin kertoo, Atollic on suunniteltu STM32-mikrokontrollereita varten, mutta toki sillä voi ohjelmoida myös suoraan PC-käyttöön tulevia sovelluksia. Atollic on Eclipse-pohjainen kuten moni muukin testattavana olleista ympäristöistä, mutta sen etuna on, että sen omistaa STMicroelectronics eli STM32 mikrokontrollereiden valmistaja, jolloin oletettavasti kyseisen mikrokontrolleriperheen tuki on taattu. Hyvänä puolena mainittakoon myös, että ympäristö on ilmainen ja se toimii Windowsilla ja Linuxilla. Ohjelmointiympäristönä Atollic sisältää kaiken tarpeellisen ohjelmiston kehittämiseen. Se sisältää siis jo itsessään kääntäjän ja virheenkorjausrajapinnan, jonka kautta myös ohjelma ladataan mikrokontrollerille. Kääntäjänä Atollic käyttää GNU gcc-kääntäjää, joka on yksi eniten käytetyistä kääntäjistä. GNU on avoimeen lähdekoodin perustuva, joten sen kehittämisestä vastaa isompi yhteisö.

ARM Keil µVision MDK

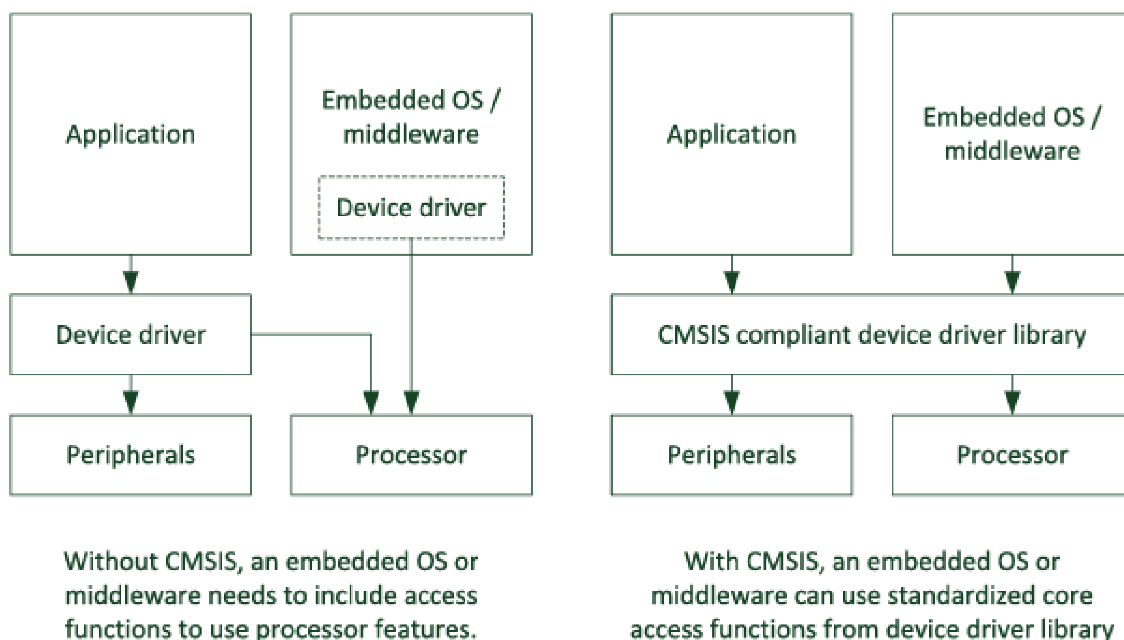
Keil on ARM:n alainen yritys, joka valmistaa ARM-laitteiden kehittämiseen tarkoitettuja työkaluja. Yksi työkaluista on Keil MDK, joka on ohjelmointiympäristö mikrokontrollereille. Se toimii ainoastaan Windows-käyttöjärjestelmässä. Se sisältää Keil:n kehittämän µVision IDE:n, virheenkorjausrajapinnan, ohjelmiston latauksen mikrokontrollerille ja ARM:n kehittämän kääntäjän. Ympäristöllä on tuki yli 5000 mikrokontrollerille, joten se ei ole sidottu pelkästään yhden valmistajan kehittämään mikrokontrolleriin. Ympäristöstä on saatavilla ilmainen versio, mutta siinä on asetettu käännettävälle ohjelmistolle maksimikoko, joten testaukseen tarvitaan maksullinen versio. Etuna tässä ohjelmointiympäristössä on todella hyvä tuki usealle mikrokontrollerille ja ison yrityksen tuki taustalla, jolloin oletettavaa on, että tuki pysyy ja kehittyy ARM-arkkitehtuuria käyttäville laitteille. Toinen valintaan vaikuttava suuri etu tässä ympäristössä on kohdeyrityksen sisäinen osaaminen ja tieto tästä ympäristöstä.

6.3.2 Ohjelmiston ja laitteiston rajapinta

Ohjelmiston ja laitteen välisenä rajapintana ARM-prosessoreissa hyödyllisintä on käyttää CMSIS-rajapintaa. CMSIS:n avulla ARM pyrkii luomaan standardin Cortex-mikrokontrollereiden ja ohjelmistojen välille, mikä parantaa ohjelmistojen skaalautuvuutta ja uudelleenkäyttöä. Otetaan esimerkkinä sulautettu ohjelmisto, joka sisältää useita ohjelmiston komponentteja:

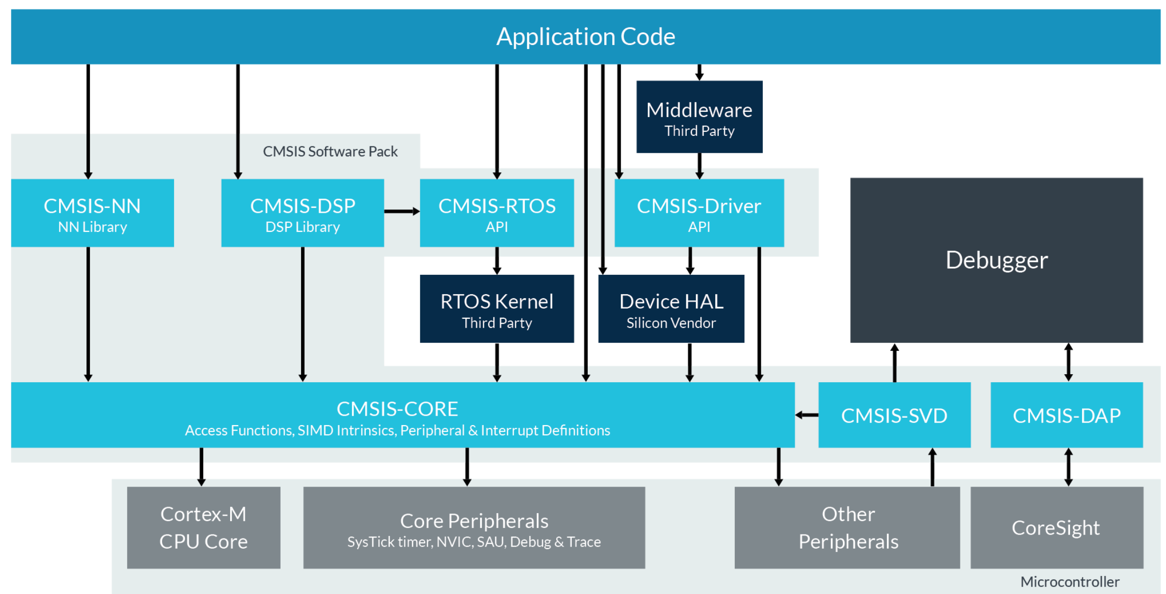
- Yrityksen sisällä kehitetty ohjelmisto
- Ohjelmisto, jota hyödynnetään aiemmista projekteista
- Mikrokontrolleritoimittajan oma rajapinta
- Mahdolliset sulautetut käyttöjärjestelmät
- Muut kolmannen osapuolen ohjelmistot, esimerkiksi tietoliikenne protokollat

CMSIS:n tarkoitus on olla riippumaton siitä, kuka mikrokontrollerin on valmistanut tai mitä ohjelmointiympäristöä käytetään, jolloin pystyttäisiin hävittämään tai ainakin vähentämään useamman ohjelmistokomponentin välisiä rajapintoja. Kuten yllä mainitussa esimerkkiohjelmistossa on useita komponentteja, on CMSIS:n tarkoitus hälventää näiden eroja, jolloin ohjelmiston uudelleen käyttö olisi huomattavasti helpompaa seuraavissa projekteissa. Kuvassa 23 on esitelty havainnoiva esimerkki CMSIS:n tarkoituksesta.



Kuva 23. CMSIS toiminta [10]

CMSIS:ssä on huomioitava, että se on Cortex-M-prosessoriarkkitehtuurin omaaville mikrokontrollereille suunniteltu, joten tarvitaan myös rajapinta, jolla voidaan käyttää mikrokontrollerikohtaisia ominaisuuksia, jos halutaan ohjelmointityö pitää yksinkertaisena ja tehokkaana. STMicroelectronics ylläpitää omaa HAL-rajapintaansa STM32-mikrokontrollereille, eli se on yrityksen ylläpitämä. Tällaiset rajapinnat voivat joskus olla jonkin yhteisön ylläpitämiä, jolloin ne ovat niin sanottuun avoimeen lähdekoodin perustuvia. Kuvassa 24 on esitelty CMSIS- ja HAL-periaate.



Kuva 24. CMSIS- ja HAL-periaate [26]

Ohjelmistonsuunnittelu tulisi tehdä niin, että sellaiset osat, jotka halutaan skaalautuvan ja ohjelmiston mahdollisesti siirtyvän toiseen Cortex-M-pohjaiseen laitteeseen, tulisi ohjelmoida CMSIS-rajapintaa käyttäen. Laitekohtaista rajapintaa eli tässä tapauksessa STM32:n rajapintaa käytettäisiin vain sellaisissa tapauksissa, joiden tiedetään olevan laitekohtaisia.

6.3.3 Versionhallinta

Kajaanissa käytetty toimintamalli on, että analysaattoreiden ohjelmistokoodi on talon sisällä yhteistä ja näkyvää. Kaikki kehittäjät näkevät myös toisissa projekteissa tehdyn työn. Pyrkimyksenä on käyttää samaa koodia ja kirjastoja kaikkialla ja tekemään eri projekteissa samankaltaisia ratkaisuja. Esimerkiksi eri projektit on organisoitu hakemistorakenteeltaan, tiedostonimiltään ja toiminnaltaan samankaltaisiksi. Tämä madaltaa ohjelmoijan kynnystä siirtyä projektista toiseen tai auttaa etsimään virheitä myös toisen tekemästä koodista.

Käytössä on Apache subversion (SVN)-versionhallintajärjestelmä, joka tarjoaa hyvän toiminnallisuuden ja on tuttu koko kehitysporukalle. GIT-versionhallintajärjestelmä tarjoaisi joissain asioissa etuja, mutta niin pieniä, että siirtymistä siihen ei toistaiseksi ole koettu perustelluksi. Versionhallinnan asiakasohjelmana käytetään TortoiseSVN-ohjelmaa, joka asennetaan kehitysympäristöön. TortoiseSVN on ollut kohdeyrityksessä käytössä jo pitkään ja on toiminut moitteetta.

6.3.4 Tietoliikenne

Tietoliikenteenä IO-kortin ja analysaattorin päätietokoneen välillä käytetään Ethernetia, johon on implementoitu Kajaanissa kehitetty comlibs-kirjasto. Tämä on Kajaanissa jo kohta kymmenen vuotta käytössä ollut kirjasto, joka tarjoaa modbus- ja eventcom-protokollat sekä TCP- että sarjaliikennemuunnelmina. Tähän liittyy myös discovery-protokolla, jolla IO-laitteet ja analysaattorit voi hakea dynaamisesti verkkosegmentistä. Kyseinen kirjasto on standardi C-koodia ja on suhteellisen helposti siirrettävissä eri ympäristöihin. Huonona puolena siinä voi pitää sitä, että liikenteen nopeutta on optimoitu niin pitkälle, että kirjaston sisäinen rakenne on ehkä tarpeettomankin monimutkainen. Tämän kirjaston vaihtaminen olisi kuitenkin käytännössä liian työlästä, koska eventcom-protokolla, joka yleensä on käytössä sen tehokkuuden takia, on jo niin pitkälle optimoitu. Sen uudelleentoteuttaminen olisi työlästä ja korvaus toisella protokollalla rikkoo IO-korttien yhteensopivuudet. Perusidea on, että IO-moduulilla on modbus-rekisterirakenteen tyylinen rekisterikartta, yksi rekisteri on 16 bittiä. Analysaattorin päätietokoneessa on vastaava rekisterikartta jokaista IO-moduulia varten. Eventcom-protokolla pitää nämä kaksi rekisterikarttaa synkronoituna keskenään ja vain tilojen muutokset lähetetään.

IO-alustassa käytettävän Ethernet-piirin käyttämiseen Wiznet on julkaisut oman kirjastonsa GitHub-verkkopalvelussa. Kirjaston on tarkoitus toimia rajapintana Ethernet-piirin ja mikrokontrollerin välillä. Kirjaston toiminta perustuu prosessien välisiin kommunikointeihin pistokkeiden (engl. socket) avulla, joissa pistoke toimii rajapinnassa tiedostokahvana, jonka kautta tietoa lähetetään ja vastaanotetaan.

6.3.5 PC-simulaatio

Testaukseen valituista ohjelmointiympäristöitä Keil µVision tarjoaa myös simulointimahdollisuuden Cortex-M-mikrokontrollereille, kun taas Atollic truestudio ei tarjoa simulointimahdollisuutta

ollenkaan. Keil:ssä simuloinnin rajoitteena on se, ettei välttämättä kaikkia mikrokontrollerin valmistajan lisäämiä ominaisuuksia voida simuloida.

Kajaanissa on aikaisemmin suunniteltu simulaattoreita itse useille eri käyttöjärjestelmille käyttämällä hyödyksi käyttöjärjestelmien ja laitteiden välisiä rajapintoja. Simulaattoreita on suunniteltu muun muassa Linuxille Qt creator-ohjelmaa käyttäen ja Windowsille Visual studio-ohjelmaa käyttäen. Yrityksessä olevan aikaisemman kokemuksen perusteella Qt creator- ja Visual studio-ohjelmointiympäristöistä on kehitysympäristöön järkevintä asentaa toinen, riippuen siitä, kumpi käyttöjärjestelmä on käytössä lopullisessa kehitysympäristössä.

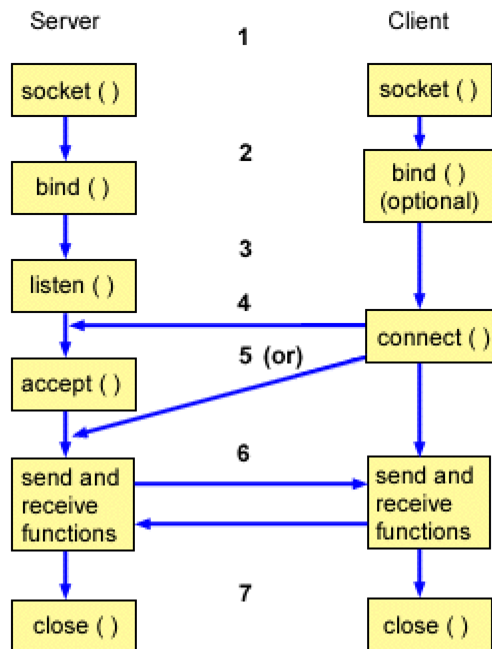
6.4 Tietoliikennekirjaston muokkaus

Ennen varsinaiseen testaukseen menemistä käytössä oleva tietoliikennekirjasto täytyy muokata toimivaksi uuteen ympäristöön. Kirjaston muokkaamisessa käytetään C-kieltä, Ethernet-ohjaimen kirjastoa ja myös tarpeen vaatiessa mikrokontrollerin rajapintakirjastoa. Tavoitteena on muokata kirjastoa mahdollisimman vähän ja niin, että kirjaston siirrettävyys ei kärsi.

Aikaisemmin mainittiin, että kirjasto on suhteellisen helposti siirrettävissä uuteen ympäristöön. Käytännössä kirjastoa täytyy muokata sen verran, että voidaan hyödyntää Ethernet-piirin TCP/IP-pinoa. Tietoliikenteen toiminta perustuu serverin ja asiakkaan välisiin pistokkeisiin (engl. socket), joiden avulla data liikkuu. Piirin kirjasto perustuu niin sanottuun Berkeley socket-rajapintaan, joka antaa pistokkeiden toiminnalle selkeän rakenteen ja syntaksin.

Pistokkeilla on tyypillisesti selkeä työnkulku, jossa serverin puolella oleva avoin pistoke odottaa pyyntöä asiakkaan puolelta. Yhteyden sitomista varten serverin on ensin määriteltävä osoite, jota kautta sen voi löytää. Kun osoite on määritelty, serveri odottaa palvelupyyntöä asiakkaalta. Yhteyden muodostamisen jälkeen voidaan aloittaa tiedonsiirto, jolloin serveri suorittaa asiakkaan lähettämän pyynnön ja vastaa asiakkaalle.

Pistokkeiden määrittämiseen ja tiedonsiirtämiseen käytetään niin sanottuja järjestelmäkutsuja, joiden syntaksi on lähes samanlainen kaikissa Berkeley-rajapintaan perustuvissa kirjastoissa. Työnkulku järjestelmäkutsuja käyttäen on esitelty kuvassa 25.

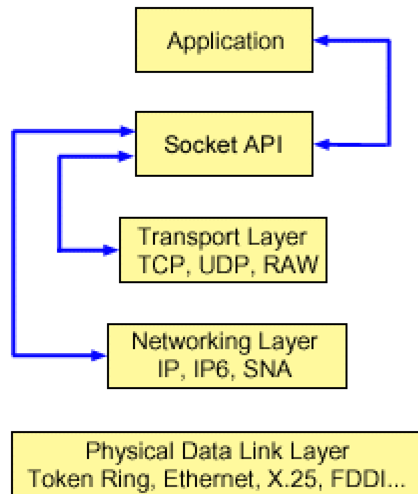


Kuva 25. Pistokkeiden työnkulku [27]

1. Serveri ja asiakas avaavat pistokkeet.
2. Serveri antaa pistokkeelle osoitteen, johon voi ottaa yhteyttä. Myös asiakas voi määrittää osoitteen, johon haluaa olla yhteydessä.
3. Serveri asettaa pistokkeen kuuntelevaan tilaan, jolloin se jää kuuntelemaan palvelupyyntöä.
4. Asiakkaan pistoke asetetaan yhteystilaan, jolloin yhteyspyyntö lähetetään.
5. Pyynnön saavuttua tehdään tarkistus siitä, hyväksytäänkö pyyntö vai ei.
6. Pyynnön hyväksymisen jälkeen voidaan aloittaa tiedonsiirto.
7. Tiedonsiirron loppumisen jälkeen suljetaan pistokkeet.

Yllä olevassa kuvassa serverin ja asiakkaan välinen tiedonsiirto tapahtuu TCP-protokollan avulla. UDP-protokolla eroaa kuvan esimerkistä siten, että niin sanotut serverin ja asiakkaan väliset kättelyt jätetään välistä, jolloin serveri saa yhteyspyynnön asiakkaalta, jonka jälkeen aloitetaan datansiirto välittämättä siitä, kuka asiakkaan puolella on.

Pistoke-rajapinta toimii siis sovelluksien välisessä kommunikoinnissa tiedonsiirtorajapintana, jolla pyritään erottamaan sovellus muista rajapinnoista. Kuvassa 26 on esitelty havainnollistava kuva, kuinka pistokerajapinta sijoittuu OSI-mallin kerroksien väliin.



Kuva 26. Pistokerajapinta OSI-mallissa [27]

Näitä periaatteita noudattaen voidaan implementoida käytössä oleva nykyinen tiedonsiirtokirjasto ja Ethernet-piirin kirjaston toiminta yhteensopiviksi.

6.5 Testaus

Kehityksen jälkeen kehitysympäristön rakenne on valmis, jolloin kahta lopulliseen testaukseen valittua ohjelmointiympäristöä voidaan testata. Testaus on helpointa suorittaa virtuaalikoneessa, jossa on käyttöjärjestelmänä Windows, koska molemmat ohjelmointiympäristöt toimivat kyseisessä käyttöjärjestelmässä. Testauksen tarkoituksena on kokeilla kehitysympäristön toimintaa ja samalla oppia siitä mahdollisia ominaisuuksia, jotka helpottaisivat muita ohjelmistonkehittäjiä tulevaisuudessa. Myös tietoliikenteen toiminnan testaus on erittäin tärkeää, jolloin saadaan selkeä kuva siitä, toimiiko tietoliikenne loogisesti ja luotettavasti. Kehitysympäristön testaukseen on hankala käyttää perinteisiä ohjelmistontestausmenetelmiä, mutta niitä voidaan hyödyntää hiukan soveltamalla. Testaus on helpointa jakaa kahteen osaan, integraatiotestaukseen ja käytettävyystestaukseen.

6.5.1 Käytettävyystestaus

Käytettävyystestauksen tarkoituksena on testata lähinnä ohjelmointiympäristöjen käytettävyyttä, koska toinen ympäristöistä valitaan käyttöön, jolla tulevaisuudessa ohjelmistot tullaan kehittämään. Käytettävyyden kannalta tärkeää on testata, kuinka testattavat ohjelmointiympäristöt toimivat, mutta myös ohjelmistojen latausta mikrokontrollerille ja tärkeimpänä ohjelmistojen debuggausta, jonka tulee toimia moitteetta. Tässä työssä käytettävyystestausta suoritetaan periaatteessa aina, kun ohjelmointiympäristöä käytetään, eli myös integraatiotestausvaiheessa. Käytettävyystestauksen tavoitteena on saada selville, miten kumpikin ohjelmointiympäristö toimii työssä käytettävän mikrokontrollerin kanssa ja oppia siitä mahdollisimman paljon, jolloin dokumentoinnin teko kehitysympäristön käyttämisestä on helpompi tehdä.

6.5.2 Integraatiotestaus

Integraatiotestauksen tarkoitus on selvittää mahdollisten komponenttien rajapintojen väliset erot eli testata toimintaa niin, että kaikki kehitysympäristön komponentit toimivat oikein. Tärkein testattava komponentti on nykyisen tietoliikenteen toiminnan varmistaminen. Tietoliikenteen toimintaa testattaessa tärkeimmiksi tekijöiksi nousee tietoliikenteen luotettava ja looginen toiminta.

7 Tulokset

Testauksen lopputuloksena saadaan selkeä kuva kehitysympäristön toiminnasta ja etenkin siitä, onko kummassakaan ohjelmointiympäristössä lopulliseen valintaan vaikuttavia tekijöitä. Testauksella haluttiin myös selvittää, toimiiko tietoliikenne oikein kirjaston muokkauksen jälkeen. Tietoliikennettä testatessa voitiin myös pinnallisesti testata tietoliikenteen luotettavuutta, mutta varmemman tuloksen tietoliikenteen luotettavuudesta saa vasta useiden pitkien testien jälkeen.

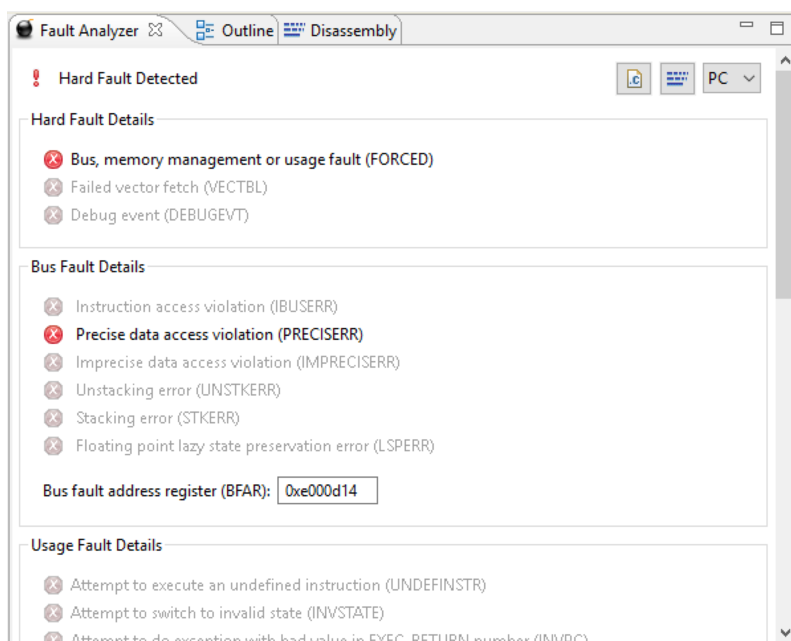
7.1 Tulosten analysointi

Tulosten analysointi jaetaan kahteen osaan, kuten testausta tehdessä, eli analysoidaan kehitysympäristön käytettävyyttä ja integraation onnistuneisuutta. Tulosten analysoinnin tarkoituksena on pohtia testauksessa syntyneitä hyviä huomioita ja mahdollisia ongelmakohtia, joita kehitysympäristössä voi olla.

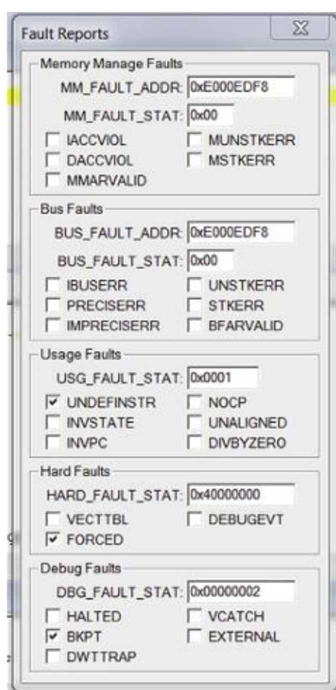
7.1.1 Käytettävyys

Käytettävyyttä arvioidessa tulisi huomioida ainakin kehitysympäristön käytön helppoutta ja sitä, kuinka helposti ja nopeasti käytön voi oppia. Testattavana olevassa kehitysympäristössä oli kaksi ohjelmointiympäristöä, jotka poikkesivat toisistaan ulkonäöllisesti ja toiminnallisesti. Molemmat ohjelmointiympäristöt tukivat kuitenkin hyvin työssä käytettävää mikrokontrolleria, mikä helpottaa huomattavasti laitteen käyttöä.

Ohjelmistojen virheenetsintä (engl. debugging) molemmissa ohjelmistoympäristöissä oli toimivaa johtuen suurimmaksi osaksi siitä, että työssä käytettävä mikrokontrolleri tarjoaa omat virherekisterinsä, joiden luku tapahtuu käytännössä samalla tavalla riippumatta siitä, mitä ohjelmointiympäristöä käytetään. Toteutustapa virheiden etsimiseen oli molemmissa ohjelmointiympäristöissä toteutettu samalla tavalla lukuun ottamatta visuaalista ilmettä, joka oli hiukan selkeämpi Atollic-ohjelmointiympäristössä. Kuvissa 27 ja 28 on esitelty esimerkit molempien ympäristöjen virheen analysoinnista virheen sattuessa.



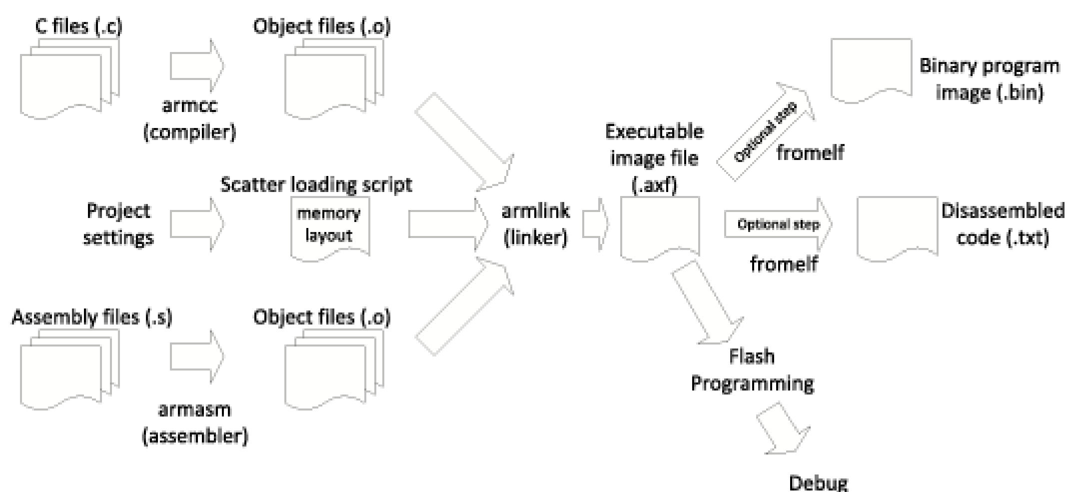
Kuva 27. Atollic virheen analysointi



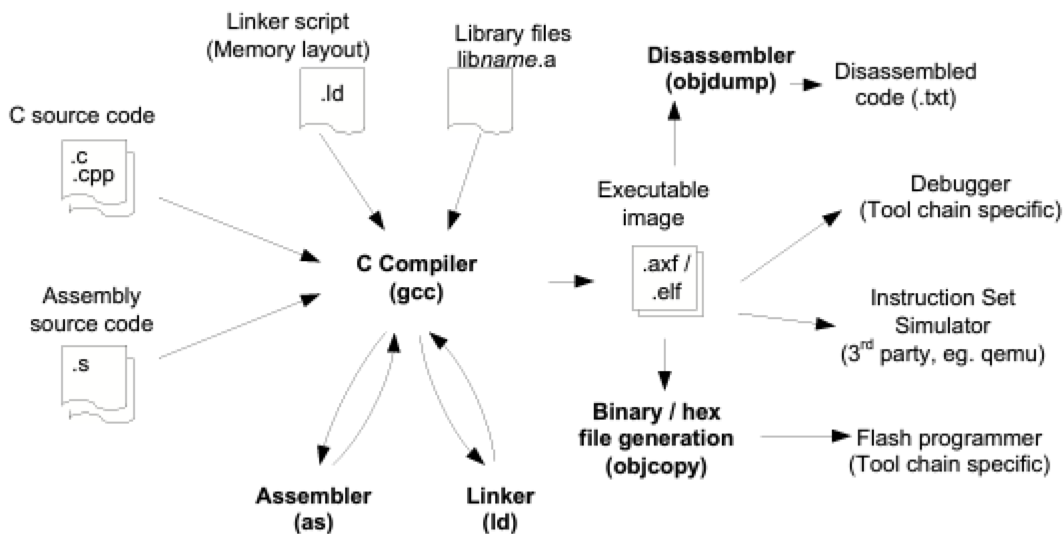
Kuva 28. Keil virheen analysointi

Kuvissa näkyy tapahtunut virhe ja mahdollisesti rekisteri, missä virhe on tapahtunut. Atollic antaa vielä mahdollisuuden tutkia koodin suoritusta, jotta voitaisiin löytää, missä virhe on tapahtunut. Atollic antaa myös selityksen virheelle, kun taas Keil käyttää lyhenteitä virheiden esittämiseen, mutta näiden lyhenteiden tarkoitus on helposti löydettävissä piirin manuaalista.

Ohjelmointiympäristöissä huomioitavaa oli myös se, että ne käyttivät eri kääntäjiä, mutta näiden kahden välillä ei käytettävyydessä juuri eroa ollut. Eroa kääntäjien välille voi syntyä niiden erilaisen toimintatavan takia esimerkiksi käännetyn ohjelman koossa. Kuvissa 29 ja 30 on esitelty esimerkki molempien kääntäjien toimintatavasta.



Kuva 29. Keilin arm-kääntäjä [10]



Kuva 30. Atollicin gcc-kääntäjä [10]

Kuvista käy ilmi molempien kääntäjien suorittama prosessi. Keil-ohjelmointiympäristö tarjoaa vielä hiukan eri vaihtoehtoja Arm-kääntäjästä, joilla pystytään saamaan ohjelmiston kokoa vieläkin pienemmäksi.

Kehitysympäristön versionhallinnassa ei ollut mitään ongelmia ja se palveli tarkoitustaan hyvin. Atollic tarjosi SVN-versionhallintaan erillisen rajapinnan suoraan ohjelmointiympäristöön, jolloin ohjelman pystyi suoraan päivittämään palvelimelle ohjelmointiympäristöstä. Myös Keil tarjosi mahdollisuuden integroida versionhallinnan suoraan ohjelmointiympäristöön, mutta tukea kohdeyrityksen käyttämälle SVN:lle ei ollut.

Ohjelmointiympäristöjä testattaessa työssä käytettiin Nucleo-kehitysalustaa, johon on integroituna ST-link-debuggeri, jonka kautta myös ohjelmistot ladataan mikrokontrollerille. Molemmissa ohjelmointiympäristöissä oli tuki tälle rajapinnalle. Debuggerin käytössä huomioitavaa on se, että monesti tietyt debuggerit avaavat paremmat mahdollisuudet ohjelmistojen debuggaukseen, mutta maksavat huomattavasti enemmän. Atollic-ympäristö ei tukenut montaa erilaista debuggerivaihtoehtoa, kun taas Keil tuki useita. Yksi Keilin tukemista debuggereista on Keilin omavalmisteinen Ulink-debuggeri, joka on ollut aiemminkin kohdeyrityksessä käytössä hyvällä kokemuksella. Testauksissa Ulink ei ollut yhtään parempi eikä huonompi kuin ST-link, mutta Ulinkillä on olemassa myös kalliimpia debuggerimalleja, joilla saavutetaan hyötyä virheenetsinnässä.

Osana kehitysympäristöä on ohjelmiston ja laitteiston väliset rajapinnat eli CMSIS ja HAL, joiden avulla testauksessa ohjelmointi suoritettiin. Ohjelmointitestauksissa kokeiltiin lähinnä piirin toimintaa kuten, kuinka väyliä ja IO-pinnejä ohjelmoidaan. CMSIS ja HAL osoittautuivat toimiviksi ratkaisuiksi ja toimivatkin moitteetta molemmissa ohjelmointiympäristöissä. Näiden rajapintojen käytössä voi ilmetä kuitenkin sellainen ongelma, että niissä on erittäin suuri oppimiskynnys. Kummankaan rajapinnan käyttö ei ole yksinkertaista ja vaatii jonkin verran perehtyneisyyttä siihen, kuinka eri ominaisuudet toimivat. Molempien rajapintojen käsikirjat ovat vaikeasti luettavia ja esimerkiksi pelkästään HAL-rajapinnan käsikirja sisältää yli 2000 sivua. Hyvä puoli näiden rajapintojen monimutkaisuudesta huolimatta on se, että se antaa ohjelmoijalle pääsyn lähelle rautaa, jolloin päästään optimoimaan ohjelmistojen ja raudan toimivuutta paremmin. Esimerkki väylän alustuksesta on esitelty kuvassa 31.

```
326 static void MX_SPI1_Init(void) {
327
328     __HAL_RCC_SPI1_CLK_ENABLE();
329
330     hspi1.Instance = SPI1;
331     hspi1.Init.Mode = SPI_MODE_MASTER;
332     hspi1.Init.Direction = SPI_DIRECTION_2LINES;
333     hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
334     hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
335     hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
336     hspi1.Init.NSS = SPI_NSS_SOFT;
337     hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_256;
338     hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
339     hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
340     hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
341     hspi1.Init.CRCPolynomial = 10;
342
343     if (HAL_SPI_Init(&hspi1) != HAL_OK)
344
345     {
346       Error_Handler();
347     }
348 }
349
```

Kuva 31. SPI-väylän alustus

Kuvasta käy ilmi SPI-väylän alustus, jonka jälkeen sitä voidaan käyttää ohjelmassa. Kuvassa oleva alustus on tehty HAL-rajapintaa käyttäen. Väylän käyttäminen vaatii aina ensin väylän alustamisen ja väylän alustamiseen tulisi ohjelmoijalla olla tieto väylän toiminnasta. Väärin alustetun väylän tai IO-pinnin takia koko ohjelma ei välttämättä toimi oikein.

Keil-ohjelmointiympäristö tarjosi myös mahdollisuuden ohjelmiston simulointiin, jota testattiin myös lyhyesti. Simuloidessa testattiin hyvin yksinkertaisia ohjelmia, mutta kuitenkin sellaisia ohjelmia, jotka sisälsivät mikrokontrollerilta vaadittavia ominaisuuksia. Simulointi vaikutti toimivalta, ja se voi helpottaa ohjelmistonkehitystyötä jatkossa. Lyhyen testauksen perusteella simulointia ei kuitenkaan voi käyttää suurissa ja vaativissa ohjelmissa, mutta pikemminkin sellaisissa tapauksissa, joissa halutaan testata jonkin suuremman ohjelman pienempää osaa.

7.1.2 Integraatio

Integraation kannalta tärkein tekijä on tietoliikenteen toimivuus kohdeympäristössä ja laitteessa. Nykyisen käytössä olevan tietoliikennekirjaston muokkaus tehtiin ensin toisella ohjelmointiympäristöllä toimivaksi, jonka jälkeen samaa muokattua ohjelmaa testattiin toisella ohjelmointiympäristöllä.

Molemmissa ohjelmointiympäristöissä muokattu tietoliikennekirjasto toimi samalla tavalla. Keil-ohjelmointiympäristössä ohjelman kääntämisen yhteydessä ilmeni joitain virheitä, jotka johtuivat siitä, että Keil-ohjelmointiympäristössä pystyy valitsemaan, millä C-kielen ohjelmointistandardilla ohjelma tarkistetaan. Tietoliikenteen toiminta ei aluksi ollut kovin loogista eikä luotettavaa, mutta kuitenkin tietoa liikkui laitteiden välillä. Haasteita kirjaston muokkauksessa aiheutti sen toiminnan saaminen loogiseksi. Useiden testauksien ja muokkauksien jälkeen päästiin kuitenkin haluttuun lopputulokseen.

Tietoliikennettä testattiin myös niin, että kokeiltiin, osaako IO-kortti ja tietoliikenne aloittaa keskustelun uudelleen sähkökatkon tai verkkoyhteyden katkeamisen jälkeen. Näiden testauksien kautta havaittiin, että IO-kortti osaa palata tällaisista vikatilanteista toimintaan. Ainut tila, mistä kortti ei osannut jatkaa enää toimintaa, saatiin aikaiseksi, kun käytettiin irti koko Ethernet-moduulia. Tästä havaittiin debuggauksen yhteydessä, että tällöin kortti menee jumiin katkenneen SPI-väylän takia. Tähän ongelmaan kuitenkin ratkaisuksi riitti muutos ohjelmaan.

7.2 Pohdinta

Tuloksista voidaan päätellä, että kehitysympäristön komponentit toimivat kuten pitää ja palvelevat käyttötarkoitustaan hyvin. Kummassakaan ohjelmointiympäristössä ei ollut mitään moitittavaa käytön kannalta ja loppujen lopuksi moni asia käytettävyydessäkin on mielipidekysymys. Käytettävyyden kannalta huomioitavaa oli myös se, että toinen ohjelmointiympäristöistä oli maksullinen, jolloin sen käyttäminen vaatii lisenssin ja lisenssiä käyttääkseen se täytyy aktivoida erikseen.

Tietoliikenteen toimivuuden kannalta Wiznetin kirjasto osoittautui toimivaksi ja oli loppujen lopuksi selkeäkäyttöinen. Testauksien aikana tietoliikenteen toiminnassa todettiin ongelmia, mutta ne johtuivat siitä, ettei kirjasto toiminut loogisesti. Tämä saatiin kuitenkin korjattua muokkaamalla kirjastoa.

Ohjelmistoja testattaessa huomattiin, että mikrokontrollerin valmistajan tarjoamasta CubeMX-ohjelmistosta oli erityisen paljon hyötyä. Tämän avulla pystyi nopeasti testaamaan haluttuja toimintoja ja ominaisuuksia. Ohjelmiston avulla pystyi myös oppimaan paljon helpommin ja nopeammin, kuinka HAL-rajapintaa käytetään.

Tulosten perusteella voidaan todeta, että uusi kehitysympäristö toimii komponentteineen ja molemmat ohjelmointiympäristöt olivat erittäin toimivia. Atollic-ohjelmointiympäristö on ilmainen, ja sillä on suuri yhteisö takana sen avoimen lähdekoodin takia. Keil on taas maksullinen, mutta kohdeyrityksessä on jo kokemusta tästä ohjelmointiympäristöstä. Se, kumpaa ohjelmointiympäristöä käytetään lopullisessa kehitysympäristössä, jää kohdeyrityksen päätettäväksi.

7.3 Jatkokehitys

Kehitysympäristö kasattiin useista eri komponenteista ja ohjelmoinneissa käytettiin monenlaisia kirjastoja. Tulevaisuudessa olisi hyvä olla myös itse tehtyjä kirjastoja, kuten HAL-rajapinnan päälle tehty kirjasto, jolla pystytään piilottamaan HAL-rajapinnan monimutkaisuus. Tällaisessa kirjastossa voitaisiin ennakkoon määritellä jo vakioidut väylät ja IO-pinnit alustuksineen, jolloin niiden käyttö itse sovelluskoodissa olisi sujuvaa ja helpompaa. Toisena lisäyksenä olisi se, että tällä hetkellä, kun kortin liittää laitteeseen, kortti ei kerro, mikä kortti se on, jolloin niin sanottu plug-and-play-asennus ei onnistu. Tämän takia olisi hyvä olla olemassa vielä ylin sovelluskerros, jossa tällaiset olisi määritelty.

Tämä työ sisälsi runsaasti tutkimista ja läheskään kaikkea tutkimuksen sisältöä ei tässä kirjallisessa työssä mainita, mutta muutamia hyviä ominaisuuksia, joita työssä käytetty piiri sisältää, voisi tulevaisuudessa harkita. Työssä käytetty piiri sisältää Flash-muistia, jonka pystyy jakamaan useampaa eri osioon, mikä mahdollistaa sen, että toista osiota voidaan ohjelmoida ja toiselta ohjelmaa voidaan suorittaa. Tästä tullaankin siihen, että myös Ethernet-verkon yli ohjelmiston lataaminen Flashille olisi erittäin kätevä.

Tulevaisuuden kannalta tärkeä asia olisi myös miettiä hiukan, kuinka seuraavalle laitesukupolvelle vaihto tullaan tekemään mahdollisimman kivuttomasti. Tämän työn yksi tarkoitus oli vähentää niitä rajapintoja, jotka voivat vaikuttaa piirin vaihtamiseen, mutta tulevaisuudessa myös piirille tehdyt ohjelmistot tulisi suunnitella niin, että ne ovat mahdollisimman helposti siirrettävissä ja jatkokäytettävissä esimerkiksi erillisten abstraktiokerrosten avulla.

8 Yhteenveto

Työn tavoitteena oli luoda uusi kehitysympäristö Valmetin analysaattoreiden uudelle IO-kortille. Työ oli voimakkaasti tutkimuspainotteinen, koska minulla ei ollut aiheesta aiempaa kokemusta. Lähtökohtana työssä oli asioiden selvittäminen ja opetteleminen. Työtä tehdessä oli myös ymmärrettävä, kuinka ohjelmistoja kehitetään järkevästi. Työn alussa voitiinkin todeta, että suurimmaksi haasteeksi muodostuu kokonaan uusien asioiden opettelu ja niiden toteuttaminen lopputulokseksi annetussa ajassa.

Työssä käytetyn piirin tutkiminen sujui hyvin siitä löytyvien dokumenttien ansiosta. Mikrokontrollerin valmistaja tarjosi myös useita esimerkkejä, joiden avulla saatiin testattua mikrokontrollerin toimintoja ja pystyttiin myös testaamaan mikrokontrollerilta toivottuja ominaisuuksia.

Käytännön toteutuksessa edettiin perinteisen ohjelmistokehitysprosessin mukaisesti. Työssä oli omat vaiheensa vaatimusmäärittelylle, suunnittelulle, kehitykselle ja testaukselle. Kehitystyössä pyrittiin hyödyntämään kestävän ohjelmistonkehityksen periaatteita: suunnitella ympäristö niin, että se on mahdollisimman helposti käytettävissä pitkälle tulevaisuuteen ja suunnitella myös ohjelmistot niin, että ne olisivat uudelleen käytettävissä. Tämä mahdollistaisi sen, että tulevaisuudessa uusilla laitesukupolvilla voitaisiin hyödyntää samoja ohjelmistoja ja vältettäisiin suurta oppimiskynnystä uusille laitesukupolville siirryttäessä.

Kehitysympäristöön asennettiin myös mikrokontrollerin valmistajan kehittämä graafinen käyttöliittymä, jolla pystyy määrittämään pinnit ja niiden halutut toiminnot. Määrittelyn jälkeen voidaan generoida valmista koodia C-kielellä. Tämä ohjelma oli suuressa osassa koko kehitystyötä, koska sen avulla pystyttiin laskemaan oppimiskynnystä huomattavasti valmiin koodin ansiosta. Tulevaisuudessa tätä ohjelmistoa voisi ainakin hyödyntää opetuksessa ja prototyyppien kanssa, mutta ei välttämättä niinkään itse ohjelmoinnissa.

Työn lopputuloksissa havaittiin, että kehitysympäristö palveli tarkoitustaan ja oli toimiva. Samalla saatiin myös toimiva tietoliikennekirjasto uudelle mikrokontrollerille ja paljon uutta tietoa mikrokontrollerin ominaisuuksista.

Lähteet

1 Valmet yrityksenä | Valmet Oy. Saatavilla: https://www.valmet.com/fi/valmet-yrityksena/valmet-lyhyesti/. Viitattu 6/11/2018, 2018.

2 Metrologia | Mittaustekniikka. Saatavilla: https://www.mikes.fi/mikes/Oppaat/metrologiasta_lyhyesti_nettiin.pdf. Viitattu 7/11/2018, 2018.

3 Koskimies K, Mikkonen T. Ohjelmistoarkkitehtuurit. Helsinki: Talentum; 2005.

4 What is a software architecture? – Definition. Saatavilla: https://www.ibm.com/developerworks/rational/library/feb06/eeles/index.html. Viitattu: 29/11/2018, 2018.

5 Haikala I, Mikkonen T. Ohjelmistotuotannon käytännöt. 12. uud. p. ed. Helsinki: Talentum; 2011.

6 Lehtonen, Tuomivaara, Rantala, Känsälä, Mäkilä, Jokela, Könnölä, Kaisti, Suomi, Isomäki & Ylitolva. Sulautettujen järjestelmien ketterä käsikirja. Turku: Painosalama; 2014.

7 Ohjelmistotuotanto, KAMK opetusmateriaali, Eero Huusko.

8 FPGAs, SoCs, Microcontrollers – A Quick Rundown of IoT Devices. Saatavilla: https://hackernoon.com/fpgas-socs-microcontrollers-a-quick-rundown-of-iot-devices-c5a25c7290c6. Viitattu 17/1/2019, 2019.

9 What is a Microprocessor. Saatavilla: https://electrosome.com/microprocessor/. Viitattu 17/1/2019, 2019.

10 Yiu J. The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors. Oxford: Elsevier Science & Technology; 2013.

11 System on a chip. Saatavilla: https://www.ee.ryerson.ca/~courses/coe838/lectures/Intro-SoC.pdf. Viitattu 17/1/2019, 2019.

12 Introduction to ARM-based System-on-Chip design. Saatavilla: http://mazsola.iit.uni-miskolc.hu/DATA/storages/files/_btXYOX__dfJfpFF.pdf. Viitattu 17/1/2019, 2019.

13 Sulautettu ohjelmointi. Saatavilla: http://www.cs.tut.fi/~sulo/pruju/pruju.pdf. Viitattu 29/11/2018, 2018.

14 Massa A, Barr M. Programming Embedded Systems, 2nd Edition. O'Reilly Media Inc; 2006.

15 Abstraction layers. Saatavilla: https://medium.com/@carlos.i.hernandez011/abstraction-layers-from-atoms-to-apps-1888e7943f14. Viitattu 28/12/2018, 2018.

16 Typical software architecture. Saatavilla: https://ebrary.net/22045/computer_science/typical_software_architecture. Viitattu 17/1/2019, 2019.

17 STMicroelectronics | ST Who We Are. Saatavilla: https://www.st.com/con tent/st_com/en/about/st_company_information/who-we-are.html. Viitattu 13/12/2018, 2018.

18 STMicroelectronics | STM32 32-bit Arm Cortex MCUs. Saatavilla: https://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html. Viitattu 13/12/2018, 2018.

19 STMicroelectronics | STM32L4 Series. Saatavilla: https://www.st.com/en/microcontrol lers/stm32l4-series.html?querycriteria=productId=SS1580. Viitattu: 13/12/2018, 2018.

20 STM32L676xx. Saatavilla: https://www.st.com/resource/en/datasheet/stm32l476je.pdf. Viitattu: 21/1/2019, 2019.

21 STMicroelectronics | HAL and LL layers. Saatavilla: https://www.st.com/content/ccc/resource/technical/document/user_manual/63/a8/8f/e3/ca/a1/4c/84/DM001731 45.pdf/files/DM00173145.pdf/jcr:content/translations/en.DM00173145.pdf. Viitattu: 21/12/2018, 2018.

22 Noviello C. Mastering STM32. Leanpub; 2018.

23 Keil | CMSIS. Saatavilla: http://www.keil.com/pack/doc/CMSIS/General/html/index.html. Viitattu 20/12/2018, 2018.

24 TCP/IP-stack. Saatavilla: https://www.hardwaresecrets.com/how-tcp-ip-protocol-works-part-1/6/. Viitattu 28/12/2018, 2018.

25 Wiznet W5500. Saatavilla: http://wizwiki.net/wiki/lib/exe/fetch.php?media=products:w5500:w5500_ds_v106e_141230.pdf. Viitattu 02/01/2019, 2019.

26 ARM | CMSIS. Saatavilla: https://developer.arm.com/embedded/cmsis. Viitattu 20/12/2018, 2018.

27 How sockets work – IBM. Saatavilla: https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_71/rzab6/howdosockets.htm. Viitattu 03/01/2019, 2019.