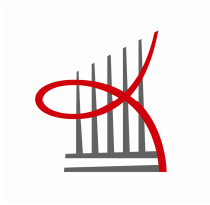


LAITEOHJELMISTON SOVITUS UUTEEN µC-ARKKITEHTUURIIN

Antti Hatunen

Opinnäytetyö
Joulukuu 2014
Tietotekniikka
Sulautetut järjestelmät ja elektroniikka

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Sulautetut järjestelmät ja elektroniikka

ANTTI HATUNEN:
Laiteohjelmiston sovitus uuteen µC-arkkitehtuuriin

Opinnäytetyö 29 sivua
Joulukuu 2014

Jokainen mikrokontrolleri tekee käytännössä samat asiat hieman eri tavalla. Esimerkiksi IIC-väyläohjain keskustelee aina standardin mukaisesti ulospäin, mutta sisäisesti ohjaus- ja datarekisterit ovat valmistajakohtaisia. Prosessorit ovat oma lukunsa – käskykannat eroavat huomattavasti valmistajien välillä. Kehittyneet kääntäjätyökalut tekevät samasta C-ohjelmakoodista toimivan binäärikoodin mille vain prosessoriperheelle, joten ongelmaksi jäävät eriävät oheislaitteet.

TreLab Oy tekee mittalaitteita teollista internetiä (IIoT) varten. Ensimmäinen tuote rakennettiin tietyn mikrokontrollerin ympärille, mutta luonnollisesti kehityksen tulee jatkua. Olisi äärimmäisen kustannustehotonta kirjoittaa käytännössä sama ohjelmakoodi uudestaan ja uudestaan jokaiselle mikrokontrolleriarkkitehtuurille. Ratkaisu ongelmaan on kerrosarkkitehtuuri, joka abstrahoi laitteistorajapinnan yleiskäyttöiseltä ohjelmakoodilta. Vain mikrokontrollerikohtainen koodi pitää sovittaa yleisten rajapintojen alatasoille.

Tässä työssä luotiin ensimmäinen prototyyppiohjelmisto ja -laitteisto uuden mikrokontrollerin ympärille. Molemmat kontrollerit olivat Atmelin valmistamia, joten Atmelin ASF-ohjelmistokehys nopeutti kehitystyötä huomattavasti. Kaikki alustustoimenpiteistä oheislaitteiden kanssa kommunikointiin ja flash-muistin käsittelyyn saatiin tehtyä nopeasti valmiilla kokonaisuuksilla. Jatkokehittely luultavasti irtaannuttaisi tehtyjä toteutuksia etenkin ASF:n ylemmiltä rajapinnoilta, mutta tämän työn tavoitteena oli tehdä mahdollisimman nopeasti toimiva prototyyppi.

Ympäröivän koodipohjan nopea kehitys aiheutti suuriakin muutoksia omaankin koodiin lähes päivittäin. Vain muutama kuukausi käytännön osuuden valmistumisen jälkeen vaille ylläpitoa jäänyt, hiomaton koodi oli jo huomattavasti päähaaraa jäljessä.

Työstä suoriuduttiin työnantajan puolesta hyväksyttävästi kohtuullisessa ajassa. Työ oli hyvä pohjatyö minkä vain mikrokontrolleriarkkitehtuurin sovittamiseen tulevaisuudessa.

Asiasanat: sulautettu järjestelmä, laiteohjelmisto, sovittaminen, ARM, TreLab Oy

ABSTRACT

Tampere University of Applied Sciences
Information Technology
Embedded Systems and Electronics

ANTTI HATUNEN:
Porting an Embedded Software to a New µC-Architecture

Bachelor's thesis 29 pages
December 2014

Every microcontroller does virtually the same thing, but in a manufacture-specific way. For example an IIC bus controller will always behave as specified in the standard, but the hardware implementation varies from vendor to vendor. The control and data registers will differ between systems. CPUs are in a class of their own with wildly varying instruction sets. However modern, advanced compiler toolchains will turn the same C program into an optimized binary for any popular architecture. Thus the multitude of peripherals poses much more of an issue.

TreLab Ltd. develops sensor solutions for the Industrial Internet of Things (IIoT). The first product was built around a specific microcontroller, but naturally there will be a change along the way. It would be highly inefficient to write practically the same program code again and again for every µC-architecture used. Using a layered software architecture mostly solves the issue. Abstracting the hardware from the application layer means that the only parts that need to be rewritten are the hardware-specific parts.

The subject of this thesis was to create the first prototype software and hardware around a new microcontroller. Both the new and old controllers were manufactured by Atmel, so the extensive Atmel Software Framework rapidly sped up development. Everything from initializing the hardware to communicating with the periphery were straightforward to develop with ready-made software components. Further development would likely bring about a separation from the ASF, especially in the higher levels. However the objective was to first make a working prototype.

The rapid development of the overlying software architecture required changes to the prototype in a day to day basis. Just a couple of months after finishing the prototype, the unmaintained and rather raw code base was already significantly behind the main branch. Long gone are the days of the waterfall model.

The prototype reached a state of approval from the employer in a reasonable time. The thesis serves as a good basis for porting any architecture in the future.

Key words: embedded system, software, porting, ARM, TreLab Ltd

SISÄLLYS

1 JOHDANTO 6
2 ARM MIKROKONTROLLEREISSA 7
2.1 SAM4L 7
2.1.1 Thumb-käskykanta 9
2.1.2 NVIC 10
2.1.3 Endianness 13
3 OHJELMISTOTYÖKALUT 14
3.1 Atmel Studio 14
3.1.1 ASF 14
3.1.2 GCC ARM toolchain ja CMSIS 15
3.2 Tuotteenhallinta 16
4 LAITTEISTO 18
4.1 Atmel SAM4L –kehitysalusta 18
4.2 Sensoritag 18
5 OHJELMISTOARKKITEHTUURI 19
5.1 Tilanne ennen 19
5.2 Kerrosarkkitehtuuri 19
6 TOTEUTUS 21
6.1 Ohjelmisto 21
6.2 Olemassaoleva laitteisto 22
6.3 Dokumentaatio ja dokumentointi 22
6.4 BLE:n liittäminen kehitysalustaan 23
6.5 Haasteet ja ongelmat 25
6.5.1 Käännösoptiot 25
6.5.2 Oman dokumentoinnin virheet 25
6.5.3 Muun koodipohjan kehittyminen 26
7 YHTEENVETO 27
LÄHTEET 28

ERITYISSANASTO

ARM	Advanced RISC Machines. 32-bittinen RISC-prosessoriarkkitehtuuri.
AVR32	Atmelin oma 32-bittinen Harvard-arkkitehtuurin RISC-prosessoriarkkitehtuuri.
Callback	Alemman rajapinnan sovellukselle välitetään parametrina osoitin, jonka osoittamaa funktiota kutsutaan tietyn ehdon täyttyessä.
Commit	Versionhallintaohjelmistoon tallennettu kokoelma koodi- ja/tai tiedostomuutoksia ohjelmistoprojektiin
CMSIS	Cortex Microcontroller Software Interface Standard – standardi, valmistajariippumaton laitteistorajapinta kaikille Cortex-M-sarjan prosessoreille.
FPU	Liukulukuyksikkö
GCC	Kokoelma vapaan lähdekoodin kääntäjiä
GIT	Linus Torvaldsin kehittämä versionhallintajärjestelmä, joka on hajautettu, eli ei vaadi aktiivista verkkoyhteyttä
Harvard-arkkitehtuuri	Prosessoriarkkitehtuuri, jossa datamuistia ja ohjelmamuistia käsitellään erillisten väylien kautta. Mahdollistaa seuraavan käskyn hakemisen ohjelmamuistista samalla, kun toinen käsky suoriutuu.
IDE	Integrated Development Environment, integroitu kehitysympäristö
IoT	Internet of Things, asioiden/esineiden internet
IIoT	Industrial Internet of Things, teollinen IoT
MPU	Memory Protection Unit, muistinsuojausyksikkö
Redmine	Selaimessa toimiva, avoimen lähdekoodin projektinhallintaohjelmisto.
RISC	Reduced Instruction Set Computer, tapa suunnitella prosessorin käskykanta mahdollisimman suppeaksi ja nopeaksi.
Toolchain	Työkaluketju: kokoelma ohjelmia, joilla voi rakentaa tuotteen.
WIC	Wakeup Interrupt Controller, sallii laitteiston aktivoimisen ulkoisesta signaalista

1 JOHDANTO

TreLab Oy [19] valmistaa mittalaitteita teollista internetiä (IIoT [9,17]) varten. Halkaisijaltaan vain hieman kahden euron kolikkoa suuremman piirilevyn ja pariston tai akun sisältävä mittalaitepaketin, *smart tagin* (kuva 1), on kyettävä toimimaan huoltovapaasti niin pitkään kuin mahdollista. Näin pieneen pakkaukseen mahtuva virransyöttöratkaisu ei luonnollisestikaan ole kovin järeä, joten jokainen mikroampeeritunti merkitsee.

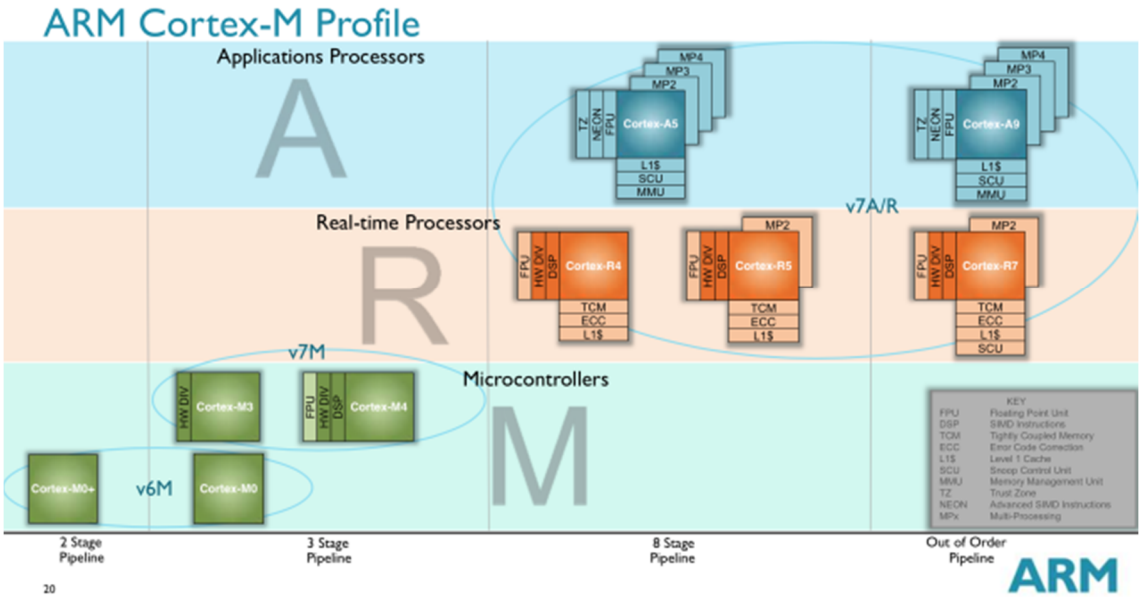
KUVA 1. TreLabin tagin kokoluokka – vas. ja kesk. prototyyppi, oik. lopullinen [19]

Mitä vaativampaa toiminnallisuutta laitteelta vaaditaan, sitä enemmän se tarvitsee laskentatehoa. Radiolinkki on ehkä suurin virtasyöppö, mutta antureita ja muita oheislaitteita hallinnoivan mikrokontrollerin virrankulutukseen on helpompi vaikuttaa - sekä ohjelmallisesti että oikean piirin valinnalla. Mitä nopeammin kontrolleri pystyy tekemään vaadittavat laskentatoimenpiteensa, sitä suuremman osan ajasta se voi nukkua, jolloin virrankulutus on minimissään.

Mittalaitteen alkuperäisenä laskentayksikkönä toimii Atmelin omaan AVR32-arkkitehtuuriin pohjautuva UC3-sarjan 32-bittinen mikrokontrolleri [3]. Mahdolliseksi seuraajaksi valikoitui ARM-arkkitehtuurin ympärille rakennettu Atmelin SAM4L-sarjan mikrokontrolleri. Siinä missä UC3-sarjalainen kykenee pienimmilläänkin aktiivisena 174 mikrowattiin megahertsiä kohden [3], SAM4L-sarjalainen voi kyetä jopa 90 µW:iin per MHz [6]. Uuteen mikrokontrolleriin ja sen myötä uuteen käskykantaan siirtyminen vaati prototyypin, jotta voitiin varmistaa edellytykset uuden tuotteen kehittämiselle. Tämän työn tarkoituksena oli prototyyppilaitteiston valmistaminen ja olemassaolevan ohjelmiston tuominen uudelle mikrokontrolleriarkkitehtuurille.

2 ARM MIKROKONTROLLEREISSA

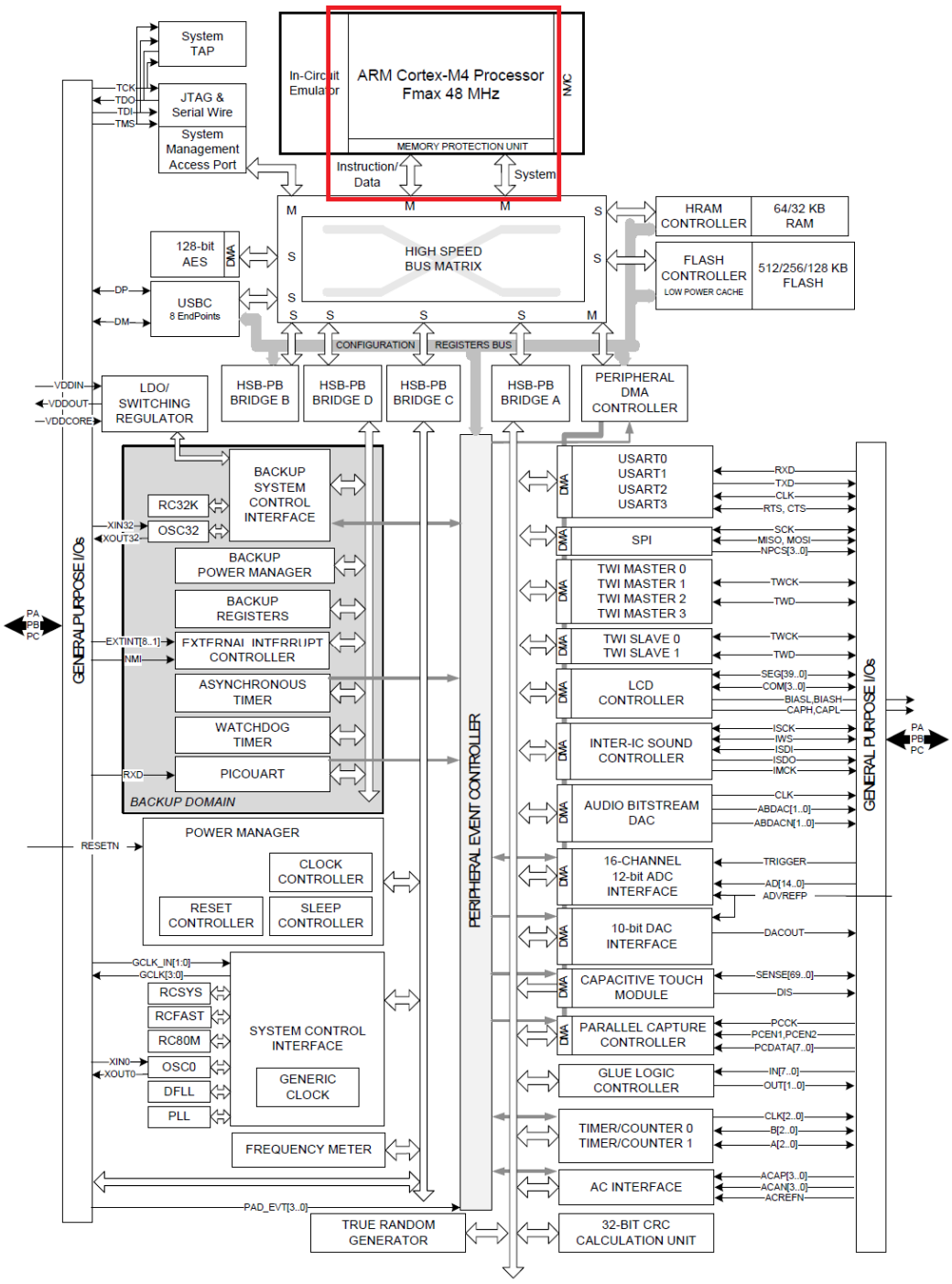
ARM on suuri tekijä nykyisten, toinen toistaan tehokkaampien, älypuhelimien vallankumouksessa. Äärimmilleen integroidut ja vähävirtaiset ARM:n Cortex-ytimiin perustuvat järjestelmäprosessorit ovat tuoneet täysiverisen tietokoneen jokaisen taskuun. Siinä missä Cortex-A -ytimiä käytetään puhelimissa ja tableteissa, todella vähävirtainen ja ominaisuuksiltaan karsittu Cortex-M on tarkoitettu TreLabin tagin kaltaisille sulautetuille järjestelmille (kuva 2). [16]



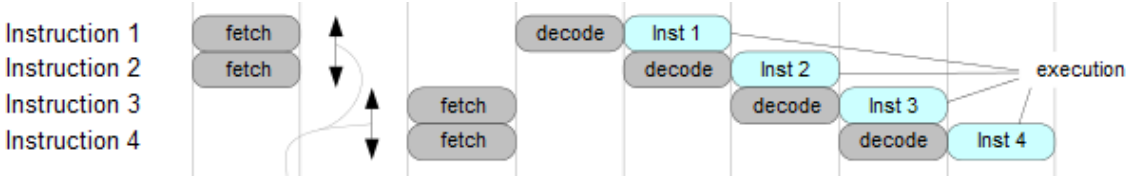
KUVA 2. ARM:n Cortex-M muihin Cortex-sarjalaisiin nähden [16]

2.1 SAM4L [6]

SAM4L on Atmelin valmistama mikrokontrolleriperhe, joka pohjautuu Cortex-M4-prosessoriytimeen varustettuna Atmelin omilla laitteistomoduuleilla (kuva 3). Cortex-M4 on Harvard-arkkitehtuurin prosessori, jossa on 3-vaiheinen liukuhihna eli suoritetaan rinnakkain käskyjen hakua, dekoodausta ja suoritusta (kuva 4).



KUVA 3. SAM4L:n arkkitehtuurilohkokaavio. Punaisella korostettu Cortex-M4:n liityntä muuhun laitteistoon [6]



KUVA 4. 3-vaiheinen liukuhihna (16-bittisiä käskyjä voi hakea kaksi kerrallaan) [1]

Cortex-M4 tukee muun muassa liukulukuyksikköä, muistinsuojausyksikköä ja ARM:in edistynyttä NVIC-keskeytysohjainta. Atmel ei ole sisällyttänyt kyseiseen kontrolleriinsa kaikkia M4:n ominaisuuksia (taulukko 1). Prosessori käyttää tiivistä Thumb-2-käskykantaa.

TAULUKKO 1. SAM4L:n Cortex-M4-implementaatio [6]

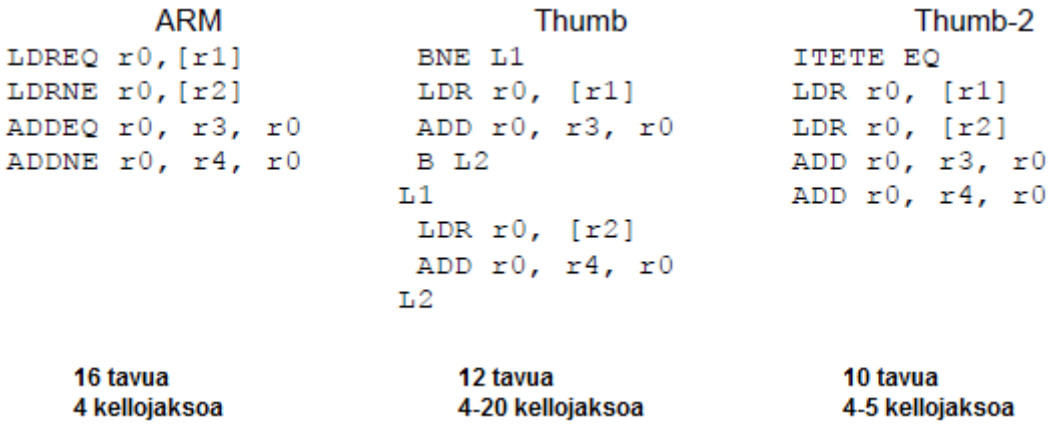
Ominaisuus	Implementaatio	Arkkitehtuurin sallima
MPU	Kyllä	
FPU	Ei	(vain Cortex-M4F:ssä)
Keskeytysten määrä	80	1-240
Prioriteettibittien määrä	4 (16 eri tasoa)	3-8 (7-255 tasoa)
WIC [1]	Ei	
Embedded Trace Macrocell [1]	Ei	
Sleep mode instruction [1]	Vain WFI [ARM]	
Endianness	Little Endian	
Bit-banding [1]	Ei	
SysTick timer [1]	Kyllä	
Rekistereiden alustusarvot [1]	Ei	

2.1.1 Thumb-käskykanta [1,2]

Thumb on osajoukko eniten käytettyjä käskyjä täydestä ARM-käskykannasta. 16-bittiset Thumb-käskyt puretaan niitä vastaaviksi 32-bittisiksi ARM-käskyiksi ennen suoritusta. Käskykanta on tarkoitettu etenkin sulautettuihin järjestelmiin, sillä Thumb-käskykannalle käännetty C-koodi vie tyypillisesti 65% ARM-käskykannalle käännetyn ohjelmakoodin viemästä kallisarvoisesta ohjelmamuistista.

Thumb-2:een on lisätty 32-bittisiä käskyjä, joita voidaan käyttää vapaasti 16-bittisten Thumb-käskyjen kanssa. Alkuperäisen Thumbin kanssa prosessorin tila tuli käskeä Thumb-moodista ARM-moodiin, jos tarvittiin täyttä käskykantaa. Thumb-2:n lisäykset tuovat koko ARM-käskykannan Thumb-tilaan, jolloin tilanvaihtoja ei tarvita.

Alkuperäisen Thumbin tiiviimpi käännöstulos ei tule ilman kompromisseja. Esimerkiksi ARM-käskykannassa jokainen käsky voi olla ehdollinen. Jokaiseen käskyyn voi siis lisätä ehdon, jonka perusteella käsky suoritetaan tai ollaan suorittamatta. Thumb-käskyissä ei 16-bittisyyden vuoksi ole mahdollisuutta ehdollisuuteen, joten samaan toiminnallisuuteen vaaditaan enemmän Thumb-käskyjä ARM-käskyillä toteutettuun koodiin verrattuna, eli aikaa kuluu lähes aina enemmän:



KUVA 5. Eri käskykantojen vertailua, ehdollisuus [13]

Huomataan, että Thumbilla on lisättävä ehdollisia hyppykäskyjä. Ehdoista riippuen suoritukseen voi mennä jopa viisinkertainen aika ARM-käskykantaan verrattuna. Thumb-2:lla on kuitenkin mahdollista lisätä käsky, joka lisää ARM-kantaa vastaavan toiminnallisuuden seuraavaan, maksimissaan neljään, Thumb-käskyyn. Saavutetaankin lähes sama suorituskyky, mutta säästetään ohjelmakoodin koossa. Thumb-2 on erinomainen käskykanta tagin käyttöön tiiviin, suorituskykyisen koodinsa ansiosta.

2.1.2 NVIC [1,6,10]

SAM4L:ssä on 80 eri keskeytystä, joista jokainen on käytännössä yksi mikrokontrollerin oheislaitteiden toiminnoista. Neljä eri prioriteettibittiä tarkoittaa, että jokaiselle keskeytykselle voi asettaa tärkeyden 16 (2^4) eri tasosta. Suurempitärkeyksinen keskeytys suoritetaan ensin. ARMin kehittynyt keskeytysohjain NVIC on lyhenne sanoista

- Nested, sisäkkäinen
- Vectored, vektoroitu
- Interrupt Controller, keskeytysohjain.

Sisäkkäisyys tarkoittaa sitä, että kun toista keskeytysaliohjelmaa suoritetaan, voi suurempiprioriteettinen keskeytys keskeyttää keskeytyksen. Näin vähäpätöisemmät keskeytykset eivät viivästytä tärkeämpien suoritusta.

Vektorointi tarkoittaa sitä, että ohjelmamuistissa on taulukko osoittimia, joista jokainen vastaa yhtä keskeytyslähdettä. Kun keskeytys tulee, keskeytysohjain hyppää lähdettä vastaavaan vektoriin ja sitä kautta keskeytysaliohjelmaan. Kuvassa 6 on esitetty pätkä CMSIS-keskeytysvektoritaulukon alustuksesta. GCC-kääntäjäkomento ”*__attribute__ ((section(”.vectors”))*” asettaa taulukon oikeaan muistiosioon eli esimääriteltyyn osoitteeseen 0x00, josta eteenpäin jokainen (osoittimen levyinen) muistiosoite vastaa yhtä keskeytyslähdettä.

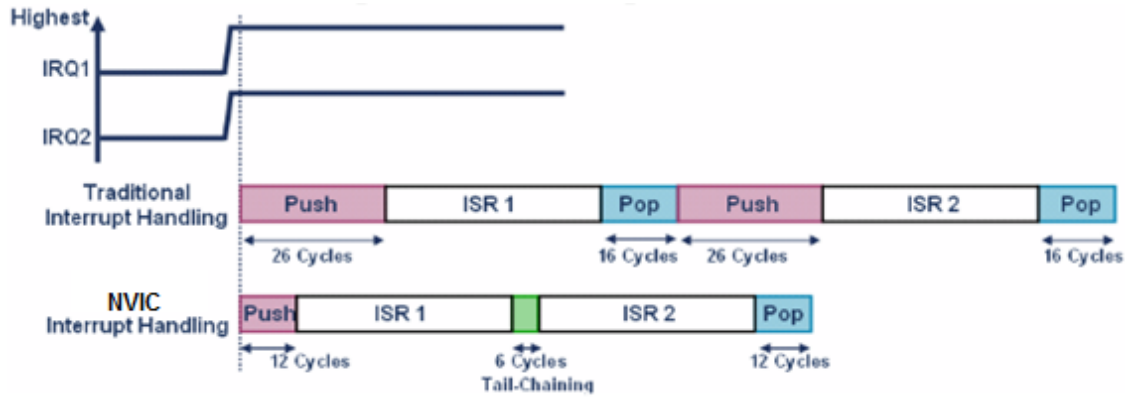
```
65   /* Exception Table */
66   __attribute__ ((section(".vectors")))
67   IntFunc exception_table[] = {
68
69       /* Configure Initial Stack Pointer, using linker-generated symbols */
70       (IntFunc) (&_estack),
71       Reset_Handler,
72
73       NMI_Handler,
74       HardFault_Handler,
75       MemManage_Handler,
76       BusFault_Handler,
77       UsageFault_Handler,
78       0, 0, 0, 0,         /* Reserved */
79       SVC_Handler,
80       DebugMon_Handler,
81       0,                  /* Reserved  */
82       PendSV_Handler,
83       SysTick_Handler,
84
85       // Configurable interrupts
86       HFLASHC_Handler,       // 0
87       PDCA_0_Handler,        // 1
```

KUVA 6. Ote ASF:n valmiista keskeytysvektorin alustuksesta

Jokainen kehittäjän konfiguroitavaksi tarkoitetun keskeytyslähteen oletuskeskeytysaliohjelma on annettu CMSIS:ssä heikkoina funktioaliaksina GCC-komennolla ”*__attribute__ ((weak, alias("Dummy_Handler")))*” [GNUMANUAL]. Funktion heikon oletustoteutuksen (alias) voi yliajaa omalla funktiototeutuksella. Esimerkiksi PDCA_0_Handler johtaisi koskemattomana loputtomaan silmukkaan CMSIS:n Dum-

my_handlerissa. Jos ohjelmistoprojektiin olisi lisätty joko itse kirjoitettu tai ASF:n PDCA-ajuri, ajurissa olisi uusi toteutus PDCA_0_Handlerille. Heikko määrittely korvaisi tällöin annetun aliaksen uudella toteutuksella. Näin on toimittava, jotta virhetilanteessa jokainen vektoritaulukon osoitin vie määriteltyyn muistiosoitteeseen.

NVIC:n nopeuden kannalta yksi tärkeimmistä ominaisuuksista on rautatason tuki ohjelman tilan tallennukselle. Kun keskeytys tapahtuu, keskeytysohjain tallentaa (push) automaattisesti suoritusaikaisen tilan (rekisterit) pinoon. Samoin keskeytyksen palvelun jälkeen laitteisto palauttaa (pop) automaattisesti pinosta keskeytystä edeltävän tilan. Keskeytysohjain osaa myös muun muassa jättää perättäisten keskeytysten välissä turhat pop-push-operaatiot pois kuvan KUVA 7 mukaisesti, sillä niillä ei olisi kuin hidastava vaikutus toimintaan. [15]



KUVA 7. NVIC:n toimintaesimerkki perinteiseen keskeytyspalveluun verrattuna, kun suoritukseen tulee samanaikaisesti kaksi keskeytystä [15]

Nopeusetu on huomattava siksi, että esimerkiksi AVR32:ssa ja monissa muissa järjestelmissä on keskeytysaliohjelman kirjoittajan huolehdittava ohjelmistotasolla tilan tallentamisessa. Push- ja pop-käskyjä tulee jokaiseen keskeytysaliohjelmaan niin monta, kuin on tallennettavia rekistereitä. GCC:n ”*__attribute__((__interrupt__))*” [10] piilottaa tämän ohjelmoijalta, eli lisää automaattisesti tarvittavat operaatiot.

Siirryttäessä AVR32-ympäristöstä ARM-ympäristöön olikin pieni kulttuurishokki, kun keskeytyskäsittelijää kirjoittaessa ei tarvinnut tehdä mitään ylimääräistä – riitti, että funktion osoite oli keskeytysvektorissa oikealla paikalla.

2.1.3 Endianness [1]

Tavujärjestys määrää tavua leveämpien tietotyyppien osatavujen arvojärjestyksen muistissa. Little endian (LE) -järjestelmässä vähiten merkitsevä tavu tallennetaan muistissa alempaan muistiosoitteeseen. Big endian (BE) -järjestelmässä vastaavasti muistissa ensimmäisenä on eniten merkitsevä tavu.

Esimerkiksi 32-bittisessä sanassa *0xDEADBEEF* tavu *0xDE* on eniten merkitsevä ja *0xEF* vähiten merkitsevä tavu. Sana tallentuu muistiin eri järjestelmissä seuraavanlaisesti:

Tallennettava heksasana:	[3]	[2]	[1]	[0]		Osoite	Big endian	Little endian
	DE	AD	BE	EF		0x00	DE	EF
	MSB			LSB	=>	0x01	AD	BE
						0x02	BE	AD
						0x03	EF	DE

KUVA 8. 32-bittinen sana eri tavujärjestyksin tallennettuna

Tavujärjestyksen merkitys korostuu siirrettäessä dataa järjestelmästä toiseen, sillä vastaanottajan on tiedettävä lähettäjän järjestelmän endianness. Tarkastellaan vaikkapa yksinkertaistettua tiedonsiirtotapahtumaa, jossa

- lähettävä järjestelmä on little endian
- vastaanottava järjestelmä on big endian
- datapaketti koostuu yhdestä 16-bittisestä puolisanasta.

Datapaketti siirretään tavu kerrallaan jotain väylää pitkin. Kun vastaanottaja on saanut molemmat tavut vastaanotettua muistiinsa, on puolitavujen paikka vielä vaihdettava, jotta arvo pysyy samana:

	Lähettäjä (LE)			Vastaanottaja (BE)			swap16()	
Muistiosoite	n	n+1		n	n+1		n	n+1
Raaka	0xAD	0xDE	->	0xAD	0xDE	->	0xDE	0xAD
uint16 tavujen merkitsevyys	[0]	[1]		[1]	[0]		[1]	[0]
Desimaaliarvo järjestelmässä	57005			44510			57005	

KUVA 9. 16-bittisen luvun siirtäminen järjestelmien välillä

3 OHJELMISTOTYÖKALUT

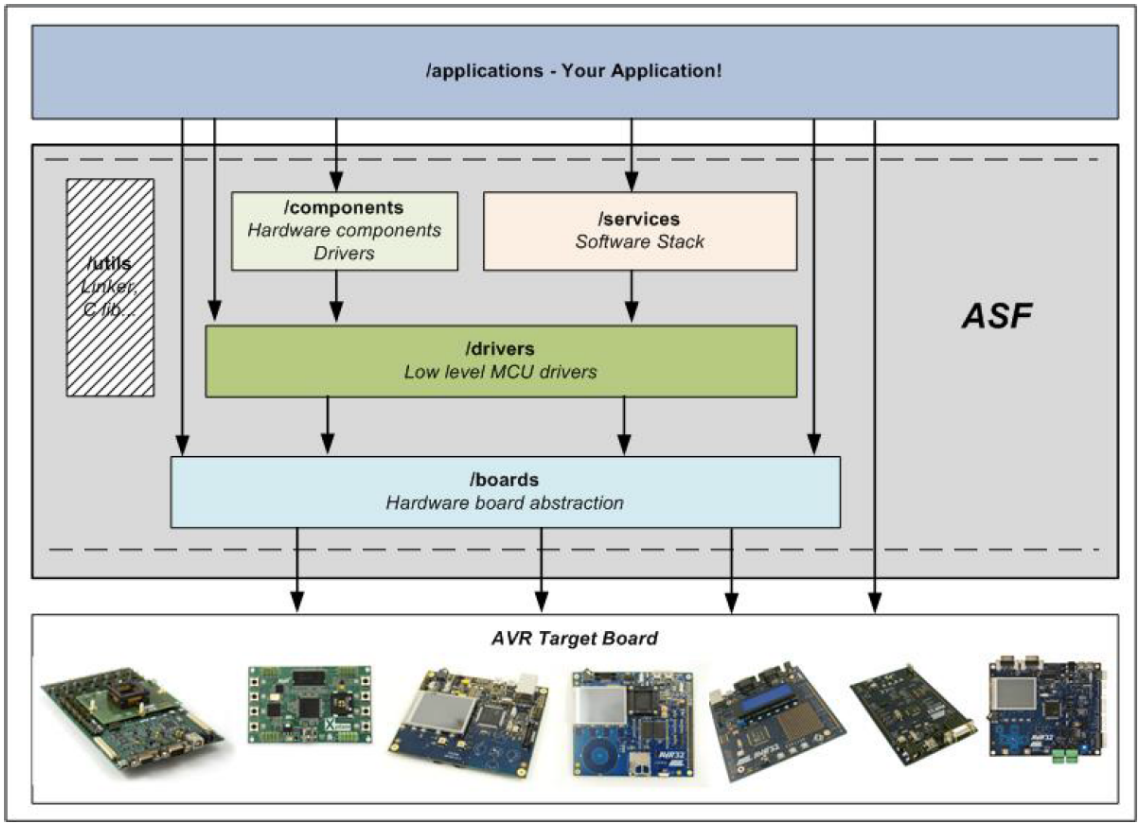
3.1 Atmel Studio [5]

Työssä käytettiin Atmelin ilmaista Atmel Studio 6:tta, joka on Visual Studio 2010:n päälle rakennettu kattava IDE. Ohjelmiston mukana tulevat muun muassa GCC:n ARM- ja AVR-variantit, laajat debug-ominaisuudet, ASF sekä WholeTomato Softwaren Visual Assist –lisenssi. Visual Assist tarjoaa työkalut koodin automaattiseen täydennykseen ja helppoon navigointiin laajankin koodin sisällä.

3.1.1 ASF [5]

Atmel Software Framework on Atmelin ylläpitämä ohjelmistokehys. Se on siis rajapinta käyttäjän ohjelmiston ja Atmelin laitteiston välillä (kuva 10). ASF:n tarkoituksena on nopeuttaa ohjelmistokehitystä, etenkin prototyyppivaiheessa, jolloin potentiaalisten asiakkaiden on helppo testata eri kontrollerien sopivuutta omiin tarkoituksiinsa. ASF sisältää ajurit ja makromäärittelyt kaikille megaAVR, AVR XMEGA, AVR UC3 ja SAM –tuoteperheiden mikrokontrollereille.

Palvelutasolla (services) ASF tarjoaa täysin samat funktioesittelyt jokaiselle tuetulle tuotteelle. Teoriassa on siis mahdollista ASF:ia käyttäen kirjoittaa yksi ohjelmakoodi, joka toimii sellaisenaan millä vain tuetulla Atmelin tuotteella. On tietysti myös mahdollista käyttää vain ajuritason (drivers) toteutuksia ja kirjoittaa välirajapinta itse, kuten pääasiassa tässä työssä tehtiin.



KUVA 10. ASF-hierarkia [5]

3.1.2 GCC ARM toolchain ja CMSIS

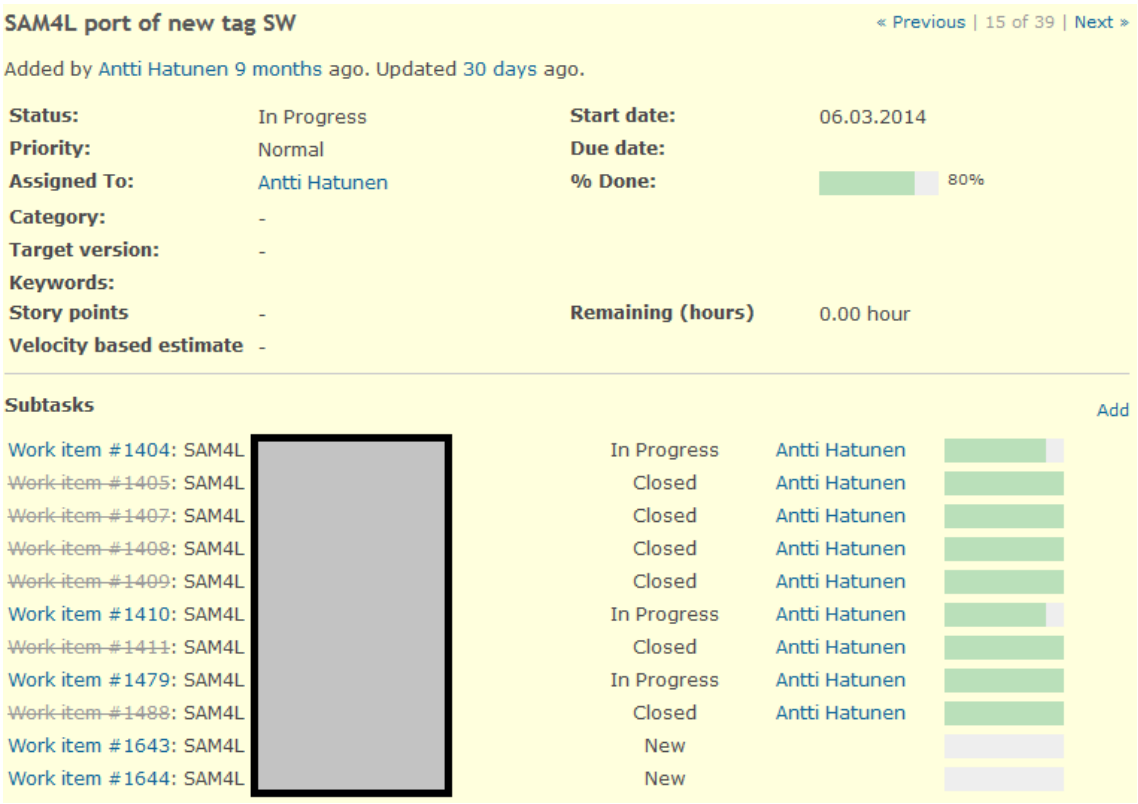
Atmel on paketoinut kehitysympäristöönsä kaikille mikrokontrollereilleen sopivan työkaluketjun, tässä tapauksessa GCC ARM toolchainin (arm-none-eabi) ja CMSIS:n. Työkaluketjussa on kaikki tarvittava, jotta C-kielisestä lähdekoodista saadaan luotua konekielinen binääritiedosto, jonka voi ladata mikrokontrollerin ohjelmamuistiin.

Atmel Studio ja CMSIS-rajapinta yhdistettynä GCC:iin ja ASF:iin poistaa kehittäjän harteilta suuren määrän pelkkää suorittavaa työtä. Kun Studiossa tekee uuden ARM-projektin, IDE tuo automaattisesti kaikki tarvittavat rajapinnat laitteiston helppoon käyttöönottoon. ASF-komponenttien tuominen poistaa vielä lisää niin sanottua turhaa työtä, jolloin voidaan tehdä vain omaan tuotteeseen liittyvä sovelluskoodi.

ASF tuo kustannuksia sekä koodin kokoon että nopeuteen, sillä jokainen ASF-kerros tuo vähintään yhden funktiokutsun lisää sovelluskoodin ja laitteiston välille. Äärimmäistä suorituskykyä haettaessa kannattaa kirjoittaa omat järjestelmäajurit ASF:n yleiskäyttöisten toteutusten tilalle.

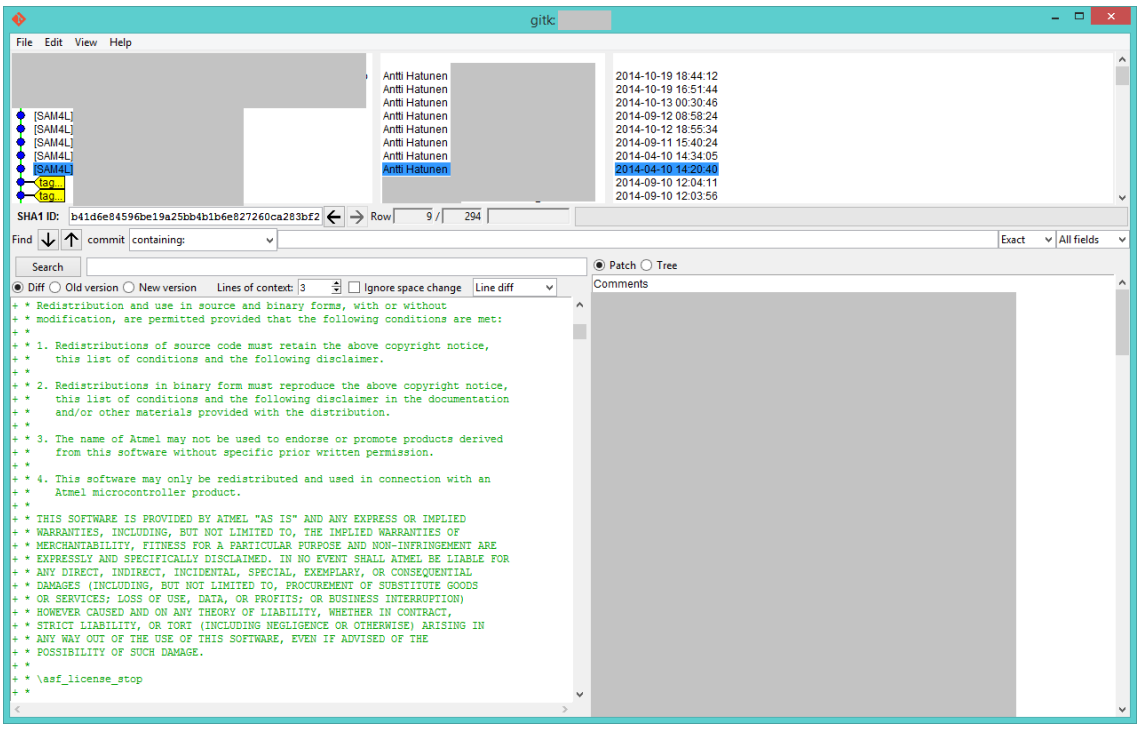
3.2 Tuotteenhallinta

Versionhallinnassa käytettiin GITiä [11] ja projektinhallinnassa Redmineä [14] (kuva 11). Työnantaja loi Redmineen työtehtävän ja osoitti sille tekijän. Työtehtävän alle tehtiin tarvittava määrä alitehtäviä, joista jokainen oli pieni perusosa koko sovelluksesta.



KUVA 11. Yleisnäkymä työkohteesta alikohteineen Redminessä

GIT piti koodilisäysten tekemisen hallinnassa, sillä muutoshistoriasta näki nopeasti (kuva 12), mitä kaikkea oli tullut muutettua. Toisaalta koodilisäysten runsauden vuoksi pieniä committeja paketoitiin ehkä turhankin tiiviiksi. Esimerkiksi yhdessä commitissa oli 56 tiedostomuutosta ja 5000 rivilisäystä. Suuret commitit vaikeuttavat myöhempää historian tarkastelua.

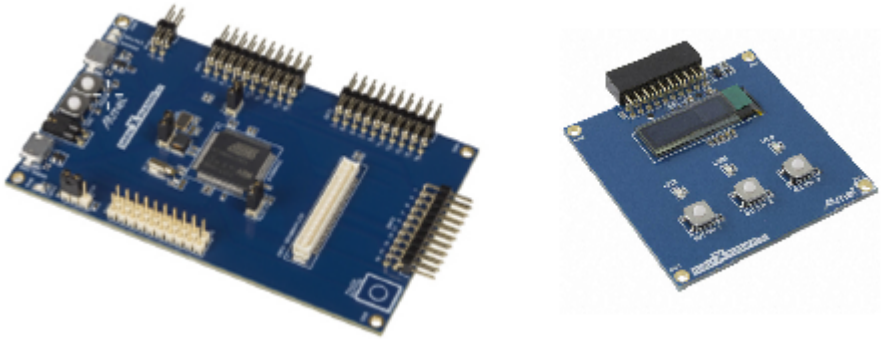


KUVA 12. GIT-repositorioselaimen (gitk) esimerkkinäkymä

4 LAITTEISTO

4.1 Atmel SAM4L –kehitysalusta

Työtä varten annettiin SAM4L Xplained Pro –kehitysalusta (kuva 13). Kaikki tarvittava oli siinä saatavilla välittömästi. Yleiskäyttöisiä protyyppejä varten olisi alustan piikkirimoihin voinut kytkeä oheislaitekortteja, kuten OLED-näytön sisältävän kortti. Kehitysalustalle oli Atmel studiossa runsaasti valmiita esimerkkiprojekteja.



KUVA 13. SAM4L Xplained Pro ja OLED-lisäkortti [7]

4.2 Sensoritag

Uusi mikrokontrolleri tuli saada kommunikoimaan alkuperäisen tagin oheislaitteiden kanssa, jotta ohjelmiston kehitykselle olisi edellytykset. Suoraviivaisinta oli juottaa kytkentälangat vanhan tagin oheislaiteväyliin ja tarpeellisiin käyttöjännitelinjoihin. Käytössä oli siis piirilevy, johon oli ladottu kaikki komponentit mikrokontrolleria lukuunottamatta.

Tagissa oli testipisteitä, joista pääsi käsiksi tiettyihin oheislaitteiden pinneihin. Kuitenkin muun muassa tiettyjen sensorien tiedonsiirtoväylään pääsi käsiksi vain väylän ylösvetovastusten kautta. 0201/0603M-kokoluokan [18] pintaliitosvastusten alle puolen millimetrin levyisiin päihin sai kerran jos toisenkin yrittää juottaa hiuksen paksuista kytkentälankaa.

5 OHJELMISTOARKKITEHTUURI

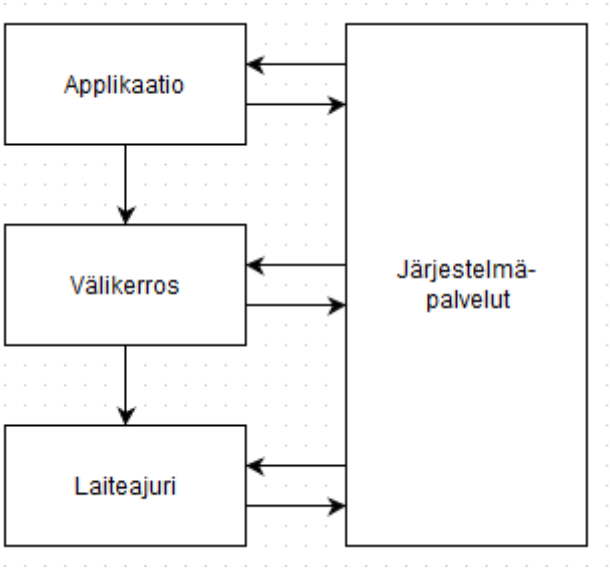
5.1 Tilanne ennen

Aina täysin uutta asiaa tehdessä toteutus on osin hakuammuntaa, vaikka sen perustaisi hyviin käytäntöihin. Todella nopean kehityksen johdosta alkuperäinen ohjelmisto palveli vain yhden mikrokontrolleriperheen tarpeita. Kehitystahti ja miehistön vähyys saivat myös oikomaan tietyissä asioissa, jolloin moduulit verkottuivat erottamattomaksi massaksi. Ajuritasolla saattoi olla hyvinkin korkean tason toimintalogiikkaa.

Alkuperäistä ohjelmistoa olisi ollut turha lähteä muovaamaan monen mikrokontrollerin tarpeisiin. Oli järkevämpää tehdä opittujen asioiden perusteella täysin uusi ohjelmisto, jonka lähtökohtana on yleiskäyttöisyys ja selkeä jaottelu. Tämän työn edellytyksenä oli uuden yleisen laiteohjelmiston kehittäminen, jonka kokeneemmat työntekijät suunnittelivat. Uuden ohjelmistoarkkitehtuurin toteutustyö alkoi hieman ennen tämän työn aloittamista.

5.2 Kerrosarkkitehtuuri [12]

Kerrosarkkitehtuuri tarkoittaa nimensä mukaisesti ohjelmistoarkkitehtuuria, joka on kerrostettu eri kokonaisuuksiin (kuva 14). Esimerkiksi applikaatiotasolla olisi sovellusohjelmia, jotka käyttävät keskitason geneerisiä palveluita, jotka taas käyttävät alimman tason komponentteja, kuten laiteajureita. Data tulee mallissa ylöspäin ja ohjaus alaspäin; ylemmän tason ohjelmistomoduulit kertovat alaspäin mitä tehdään, ja saavat erinäisin metodein takaisin haluamansa datan. Kaiken vierellä tai alla on järjestelmä, jonka palvelut, kuten ajastimet, ovat kaikkien käytössä.



KUVA 14. Kerrosarkkitehtuurin yksinkertaistettu havainnekuva

Siinä missä applikaatio on pelkkää älyä eli logiikkaa, ajuritasolla tehdään tasan tarkkaan se, mitä ylempi taso käskee. Välitasolle asti voidaan kirjoittaa täysin geneeristä koodia. Välitaso ja käytetyt järjestelmäpalvelut ovat abstrahoituna geneerisiksi funktioiksi (IIC_read yms.), jotka sitten voivat todellisuudessa käyttää laitteistosta riippuvalla tavalla todellisia oheislaitteita toiminnallisuuden aikaansaamiseksi.

Kuvatun kaltainen rakenne mahdollistaa yleisen koodin todella helpon liittämisen arkkitehtuurispesifiseen koodiin. Implementaatiokohtaisten yksityiskohtien piilottaminen ylemmiltä rajapinnoilta johtaa myös selkeämpään sovelluskoodiin. Selkeiden suhteiden määrittely kerrosten välillä luo entisestään lisäselkeyttä; alemmat rajapinnat eivät saa tietää ylemmistä mitään. Data välitetään callbackien tai parametrien välityksellä ylöspäin – aina ylemmän rajapinnan toimesta.

6 TOTEUTUS

Käytännön työtä tehtiin muun palkkatyön ohessa, siis pääasiassa päivisin. Suorituksen rikkoivat päiviksi tai jopa viikoiksi kiireelliset asiakaslähtöiset kehitys- ja bugikorjaustehtävät. Kun työnantajan mielestä palaset olivat hyväksyttävästi kohdillaan, siirtyi työhön käytetty aika palkallisen työn ulkopuolelle. Kirjallinen osuus kokonaisuudessaan ja joitain osia käytännön osuudesta tehtiin kotona. Hyvät etätyömahdollisuudet auttoivat suuresti.

6.1 Ohjelmisto

Luotiin ohjelmistoprojekti valmiin ASF:n tarjoaman SAM4L-projektin päälle, sillä se oli suoraviivaisin tie olla tekemättä turhaa työtä. Valmiit ohjelmistokomponentit lisättiin hierarkiansa mukaisesti projektiin (show all->add to project). Luotiin hierarkiaan SAM4-spesifiset polut ja tiedostot. Luotiin pinnimäärittelyt tehtävänannon mukaisesti, jotta prototyyppi vastaisi mahdollisimman tarkasti mahdollista valmista tuotetta.

Muokattiin keskitasot käyttämään SAM4-spesifisiä rajapintoja. Tehtiin toteutukset rajapinnoille pala kerrallaan. Aloitettiin ohjelmiston toiminnan ja kehityksen kannalta elintärkeistä toiminnoista, kuten GPIO-porttialustuksista, debug-liitynnän pystyttämisestä ja kellotaajuuksien asettamisesta. Edettiin moduuli kerrallaan, pyrkimyksenä saada ensin järjestelmä pystyyn, nukkumaan ja indikaattoriledi vilkkumaan debugdatan siivittämänä.

Kun laitteella oli toimintaedellytys, voitiin siirtyä sensoriajurien ja muiden toissijaisten oheislaitteiden sovittamiseen uudelle arkkitehtuurille. Luotiin liitynnät radiotielle. Tässä vaiheessa tehtiin myös fyysiset liitynnät dummytagista testilevyyn, jotta saatiin edellytys ohjelmiston testaamiselle.

Jokainen ohjelmistomoduuli pyrittiin testaamaan yksitellen sitä mukaa, kun niitä toteutettiin. Luonnollisestikaan näin ei saatu kiinni kaikkia virheitä, vaan kokonaisvaltaisemmassa järjestelmän testaamisessa tuli ilmi monia pieniä virheitä. Workflowksi muodostui toteutus-testaus-virheenkorjaus-sovitus-testaus-virheenkorjaus.

6.2 Olemassaoleva laitteisto

Alkuperäisestä tuotteesta oli saatavilla versio, johon ei ollut ladottu mikrokontrolleria. Tällaisesta ns. dummytagista oli melko helppoa johdottaa IIC-väylä ja käyttöjännitteet sekä tarvittavat radiolinkkiin liittyvät pinnit, jolloin kaikki toiminnallisuus saatiin käden käänteessä prototyypin käyttöön.

6.3 Dokumentaatio ja dokumentointi

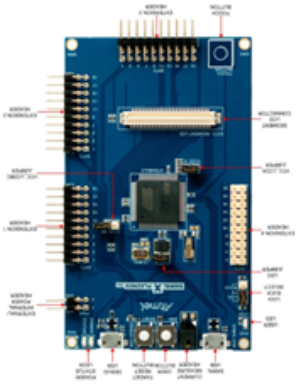
Työtä helpottamaan oli syytä tehdä dokumentaatiot tarvittavista liitynnöistä dummytagissa ja kehitysalustassa. Kehitysalustan pinnimerkinnät olivat tässä tapauksessa epäsuotuisasti piirilevyn pohjassa, ja datalehdissä tieto oli hajallaan. Tehtiinkin oma pinnikartta, josta näki helposti pinnien sijainnit (kuva 15). Karttaan merkittiin myös pinnien ohjelmoidut toiminnot, mutta ne on jätetty kuvasta pois. Kyseinen pinnikartta tehtiin vasta, kun oltiin tehty virhe ilman sitä (6.5.2).

Apuna olivat myös dummytagin ja radiomoduulin valmiit pinnikartat. Käsinkosketeltavat dokumentit olivat korvaamaton apu kytkentälankoja juotettaessa ja kytkiessä komponentteja toisiinsa. Koekytkentäjohdotukset oli välillä irroitettava, joten ne pyrittiin niputtamaan helposti uudelleen kytkettäviksi.

Työn edistyminen dokumentoitiin Redmineen ja koodimuutokset nähtiin GITistä. Päiväkohtaiset työtehtävät merkittiin myös tuntikirjanpitoon työnantajaa varten. Olennaisia asioita kirjoitettiin muistiin opinnäytetyön kirjallista osuutta varten.

EXT3									
2	4	6	8	10	12	14	16	18	20
GND	PB04	PC16	PA09	PA10	PB15	PC27	PA20	PA18	VCC
ID3	PB03	PC15	PA08	PA06	PB14	PC26	PA17	PA19	GND
1	3	5	7	9	11	13	15	17	19

EXT2			
20	VCC	GND	19
18	PC30	PA21	17
16	PA22	PB11	15
14	PC27	PC26	13
12	PB15	PB14	11
10	PC09	PC06	9
8	PC05	PC04	7
6	PB10	PC08	5
4	PB02	PA07	3
2	GND	ID2	1



EXT1			
20	VCC	GND	19
18	PC30	PA21	17
16	PA22	PC03	15
14	PB01	PB00	13
12	PA24	PA23	11
10	PB13	PC25	9
8	PC01	PC00	7
6	PC02	PB12	5
4	PA05	PA04	3
2	GND	ID1	1

EXT4			
20	VCC	GND	19
18	PA18	PA19	17
16	PA20	PA11	15
14	PB01	PB00	13
12	PB15	PB14	11
10	PA15	PA16	9
8	PA13	PA12	7
6	PC18	PC02	5
4	PC07	PB05	3
2	GND	ID4	1

KUVA 15. SAM4L Xplained Pro:n värikoodattu pinnikartta [7]

6.4 BLE:n liittäminen kehitysalustaan

Radiomoduulin ja mikrokontrollerin ohjelmallisessa yhdistämisessä tuli ensimmäistä kertaa eteen eri tavujärjestykset eri järjestelmissä. Radio on Little endian ja AVR32 Big endian [3], ja ennen tavujärjestys oli sovitettu ns. kovakoodattuna. Sen sijaan SAM4 on Little endian [6], joka tuotti odottamattomia ongelmia (KUVA 9).

Suurin osa radioliikenteeseen liittyvästä kommunikaatiosta on tavumittaista, joten tiettyyn pisteeseen asti ohjelmisto toimi oikein. Ensimmäisen 16-bittisen arvon kohdalla tuli kuitenkin virhe, sillä oikean sanan tavut olivat tietysti väärässä järjestyksessä. Debug-tulosteilla virheen jäljlle päästiin onneksi melko nopeasti.

Työn ohessa tulikin ajankohtaiseksi implementoida tavujärjestyksen vaihto tarpeellisiin kohtiin ohjelmistossa. Radion ja SAM4:n tai muun Little endian -arkkitehtuurin kanssa tavujärjestystä ei tarvitse vaihtaa, mutta UC3:n tai muun Big endianin kanssa tarvitsee. Kuitenkin saman ohjelmakoodin on toimittava kaikilla arkkitehtuureilla. C-kielessä tämä on suoraviivaista toteuttaa makroilla, esimerkiksi:

```
#ifdef AVR32
/* AVR32 ’host’ on Big endian, vaihda sanan puolikkaat keskenään */
#define LE_TO_HOST_16(x) ((((x) << 8) & 0xFF00) | (((x) >> 8) & 0x00FF))
#elif ARM
/* Molemmat Little endian, pidä arvo ennallaan */
#define LE_TO_HOST_16(x) (x)
#endif
```

Kyseisen kaltaiset makrot tehtiin, mutta sitten huomattiin ASF:n tarjoaman compiler.h –tiedoston valmiit makrot. Kyseinen header-tiedosto sisältää aina käytössä olevalle ASF-arkkitehtuurille sopivat makrot tavujärjestyksen vaihtoon. Lisäksi jos mikrokontrollerissa on käsky tavujen vaihtamiseen, voidaan käyttää kyseistä käskyä (ns. inbuilt-toiminnallisuus), mutta muuten tavut vaihdetaan esimerkin tapaisesti.

Ilman valmiita makroja olisi tullut huomata AVR32-käskykannan tarjoamat swap-käskyt, sillä GCC ei osaa käyttää niitä käännöstuloksessaan ilman ohjelmoijan toimenpiteitä, eli

1. GCC:n __builtin_bswap_16 tai __builtin_bswap_32:n käyttöä [8] tai
2. oman assembly-koodin kirjoittamista inline-assemblynä.

6.5 Haasteet ja ongelmat

6.5.1 Käännösoptiot

Pienen tauon jälkeen palattiin taas työn pariin. Ennen taukoa I2C-kommunikointi toimi ja sensoreiden kanssa ei ollut ongelmaa. Nyt I2C ei yhtäkkiä toiminutkaan. Debug-istunnolla selvisi, että ASF:n ajurin sisällä jäätiin loputtomaan silmukkaan. Tutkittiin, mitä muutoksia oli ehditty tekemään. Lopulta selvisi, että GCC:n optimointitasot 1-3 (-O[n]) aiheuttivat ilmiön. Koodin koon optimoiva –Os ja ennen päällä ollut ei optimointeja (-O0) eivät aiheuttaneet jumiutumista. [10]

Keskustelupalstoilta tai Atmelin bugiraporteista [4] ei löytynyt vastaavaa ongelmaa. Käännöstulosta tutkimalla ei nähty mitään selvää syytä toiminnalle koodin puolesta, joten ilmeisesti moduulin konfiguroinnissa tapahtui jotain erilailla. Oskilloskooppimittaus näytti, että kellosignaali vaihtoi tilaa, mutta linja jäi muuten hiljaiseksi. Ei nähty syytä käyttää enempää aikaa vianetsintään, sillä kyseisiä optimointitasoja ei tarvittu. Ahkerampi yhteisön jäsen olisi käyttänyt hieman aikaa bugiraportin kirjoittamiseen.

6.5.2 Oman dokumentoinnin virheet

Työnteon helpottamiseksi tehtiin pinnikartta kehitysalustasta, johon merkattiin käytetyt pinnit ja niiden funktiot. Data- ja ohjekirjoja lukiessa tapahtui kuitenkin aluksi virhe. Kehitysalustan 90 asteen kulmaan käännetyt piikkirimat ovat toiminnoiltaan identtiset, joten lukuvirhe sijoitti omassa dokumentaatiossa väärät pinnit väärään piikkirimaan.

Tästä juontui melko pitkä vianetsintä, sillä kyseiset pinnit olivat radion ja mikrokontrollerin välisen sarjaliikenneväylän kättelypinnit. Pinnit johdotettiin siis radiomoduulista käytöstä pois, suuri-impedanssisessa tilassa, oleviin mikrokontrollerin pinneihin. Osaltaan haastavat johdotukset saivat ensin epäilemään juotosvikaa, joten osana tikanheittotyylistä vianetsintää käänneltiin ja paineltiin johtoja samalla kun tarkistettiin, että ne olivat dokumentaation mukaan oikeassa paikassa.

Kytkennän sörkkiminen käsin johtikin siihen, että kättelypinneihin indusoitui jännite, jonka osapuolet havaitsivat liikenteen aloittamiseksi! Debug-tulosteisiin ilmestyi satunnaisen epäsatunnaisesti liikennettä, joka sai vianetsinnän väärille urille. Vasta turhauttavien ohjelmakoodinlukuhetkien ja oljenkorsien havittelemisien jälkeen rauhoituttiin ja katsottiin vielä kerran, että kaikki on kytketty oikein.

6.5.3 Muun koodipohjan kehittyminen

Työtä tehtäessä muu koodipohja oli arkkitehtuurimuutoksen vuoksi vielä muotoutumassa. Saattoi olla viikkoja, jolloin koodiin tuuli muutoksia joka päivä, ja oma työ tuli sovittaa muun ohjelmiston kehitykseen. Muiden työtehtävien jälkeen saattoikin kulua kokonainen työpäivä vain siihen, että päivitti omat moduulinsa ylempien tasojen kanssa yhteensopivaksi.

Tämän työn kirjoittamisen aloittaminen tapahtui vasta useita kuukausia koodin hyväksyttävän tilan saavuttamisen jälkeen. Tänä aikana muu koodipohja olikin kehittynyt huimaa vauhtia, ja SAM:n ajurit tuli liittää uusiin ominaisuuksiin. Olisi siis ollut järkevää kehittää myös SAM-haaraa päähaaran ohessa, mutta tuotehallinnassa pääpaino oli alkuperäisen mikrokontrolleriarkkitehtuurin saaminen tuotekelpoiseksi. Aktiivisuutta olisi voinut löytyä omasta puolesta enemmän.

7 YHTEENVETO

Tavoitteena oli tehdä toimiva prototyyppilaitteisto uudelle laitearkkitehtuurille. Uusi käskykanta ja mikrokontrolleri oli sovitettava olemassaolevaan laiteohjelmistoon. Tässä tavoitteessa onnistuttiin halutussa mittakaavassa.

Työ haastoi toden teolla, sillä lähes kaikki oli uutta. Oman mausteensa toi todella kiivas kehitystahti koko ohjelmistossa. Oli tehtävä jatkuvasti korjausliikkeitä ja mukauduttava jokapäiväisiin, usein melko suuriinkin, koodimuutoksiin. Muut työtehtävät rikkoivat osaltaan työn suorittamista, sillä aina työn pariin palatessa oli orientoiduttava uudestaan siihen, mitä oikein olikaan tekemässä.

Tämän työn yhteydessä oli ilo huomata, että kaikki TAMK:ssa opetettu tuli tarpeeseen, etenkin elektroniikan ja sulautettujen järjestelmien osalta. Sama havaittiin myös edelliskesän työharjoittelussa, johon lähdettiin ummikkona. Koulussa opitut asiat, etenkin oppimaan oppiminen, antoivat eväät työstä suoriutumiseen ja itsekehitykseen.

Heikkoudet työssä olivat kirjallisen työn valmistuminen ja kommunikaatio. Kirjallinen työ ja sitä myöten valmistuminen saatiin pakettiin puoli vuotta suunniteltua myöhemmin, vaikka kunnialliseen suoritukseen oli kaikki edellytykset.

Siirtyminen AVR32-arkkitehtuurista ARM-arkkitehtuuriin antaa hyvän pohjan tuleville sovitustöille. Työn suurin anti oli näyttö siitä, että laiteohjelmistoa voidaan käyttää missä vain ympäristössä, käskykannassa tai mikrokontrollerissa.

LÄHTEET

1 ARM. 2014. Cortex-M4 Processors. Ohjekirja. Vierailtu 4.10.2014. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.cortexm.m4/index.html

2 ARM. 2010. Cortex-M4 Devices. Generic User Guide. Käyttöopas. Luettu 3.12.2014. http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DUI0553A_cortex_m4_dgug.pdf

3 Atmel. 2014. 32-bit AVR UC3 Microcontrollers. Tuotesivu. Luettu 27.9.2014. http://www.atmel.com/products/microcontrollers/avr/32-bitavruc3.aspx

4 Atmel. 2014. Atmel Software Framework Bug Tracker. Vierailtu 18.6.2014. http://asf.atmel.com/bugzilla/

5 Atmel. 2012. Atmel Software Framework - Reference Manual. Application note. http://www.atmel.com/images/doc8432.pdf

6 Atmel. 2014. ATSAM ARM-based Flash MCU, SAM4L Series. Datalehti. Luettu 27.9.2014. whttp://www.atmel.com/Images/Atmel-42023-ARM-Microcontroller-ATSAM4L-Low-Power-LCD_Datasheet.pdf

7 Atmel. 2013. Atmel SAM4L Xplained Pro. Käyttöohje. http://www.atmel.com/Images/Atmel-42074-SAM4L-Xplained-Pro_User-Guide.pdf

8 Atmel. 2007. Getting started with GCC for AVR32. Application note. Luettu 13.11.2014. http://www.atmel.com/Images/doc32074.pdf

9 Fleisch, Elgar. 2010. What is the Internet of Things? White paper. http://cocoa.ethz.ch/media/documents/2014/06/archive/AUTOIDLABS-WP-BIZAPP-53.pdf

10 Free Software Foundation. 2014. A GNU Manual. Ohjekirja. Vierailtu 2.12.2014. https://gcc.gnu.org/onlinedocs/gcc/

11 Git. 2014. Git. Kotisivu. http://git-scm.com/

12 Laine, Harri. 2001. Ohjelmistoarkkitehtuurit. Kerrosarkkitehtuuri. HY/TKTL. Luentokalvosarja. Luettu 3.12.2014. https://www.cs.helsinki.fi/u/laine/arkki/k01/ark01_2.pdf

13 Phelan, Richard. 2003. Improving ARM Code Density and Performance. ARM. Luettu 3.12.2014. http://www.cs.uiuc.edu/class/fa05/cs433ug/PROCESSORS/Thumb2.pdf

14 Redmine. 2014. Redmine. Kotisivu. Vierailtu 1.12.2014. http://www.redmine.org/

15 Sadasivan, Shyam. 2006. An Introduction to the ARM Cortex-M3 Processor. ARM. White paper. Luettu 2.12.2014. http://www.arm.com/files/pdf/introToCortex-M3.pdf

16 Shimpi, Anand. 2014. ARM's Cortex M: Even Smaller and Lower Power CPU Cores. AnandTech. Luettu 27.9.2014. http://www.anandtech.com/show/8400/arms-cortex-m-even-smaller-and-lower-power-cpu-cores

17 Teich, Paul. 2013. Connecting with the Industrial Internet of Things (IIoT). Moor Insights & Strategy. Luettu 1.12.2014. http://www.moorinsightsstrategy.com/wp-content/uploads/2013/10/Connecting-with-the-Industrial-Internet-of-Things-IIoT-by-Moor-Insights-Strategy.pdf

18 Topline. 2014. Size Selection Chart. Vierailtu 2.12.2014. http://www.topline.tv/SizeChart.html

19 TreLab Oy. 2014. Yrityksen kotisivut. Luettu 27.9.2014. http://www.trelab.fi/