Tampereen ammattikorkeakoulu

Ohjelmiston vianetsintä sulautetuissa järjestelmissä

Mikko Rauman

OPINNÄYTETYÖ
Marraskuu 2019

Tietojenkäsittelyn koulutus
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

RAUMAN, MIKKO:
Ohjelmiston vianetsintä sulautetuissa järjestelmissä

Opinnäytetyö 32 sivua
Marraskuu 2019

Opinnäytetyön toimeksiantaja oli tamperelainen UnSeen Technologies Oy. UnSeenillä oli aloitettu uusi projekti, joka perustui sulautettuihin järjestelmiin. Uuten projektiin ilmeni tarve kouluttaa henkilökuntaa vianetsintään sulautetuissa järjestelmissä.

Tavoitteena opinnäytetyössä oli lisätä UnSeenin nykyisten ja tulevien sovelluskehittäjien tietotaitoa sulautettujen järjestelmien vianhausta sekä tutkia ja verrata menetelmiä, joita vianhaussa voidaan käyttää hyväksi. Tämän selvityksen pohjalta oli tarkoitus valita mahdolliset menetelmät projektiin ja luoda dokumentaatio työkalujen käyttöönottoon.

Tulokset saatiin tutkimalla sovelluskehityksen yhtäläisyyksiä ja eroavaisuuksia sulautetun ja perinteisen vianetsinnän pohjalta. Eri työkalut ja menetelmät arvioitiin projektissa tapahtuneiden ongelmatapausten pohjalta. Nämä tapaukset kuvattin esimerkkitapauksina opinnäytetyössä. Valitut menetelmät olivat ajonaikainen testaus, testivetoinen kehitys ja instrumenttipohjainen vianetsintä. Näihin jokaiseen kuvattiin myös menetelmissä käytetyt työkalut.

Sovelluksen ajonaikainen testaus on melko muuttumatonta sulautetuissa järjestelmissä. Sovelluskehittäjä voi edelleen käyttää kehitysympäristöstä tuttua vianetsintätyökalua menetelmään. Testivetoinen kehitys taas ei vielä soveltunut projektiin testattavien loogisten kokonaisuuksien vähyyden ja laitteiston asettamien rajoitteiden vuoksi. Instrumenttipohjaisessa vianetsinnässä virrankulutuksen profilointi havaittiin loogiseksi jatkeeksi vianetsintään. Lisäksi tuloksena syntyi UnSeenin omaan verkkoon ohjeet työkalujen käyttöönottoon.

Asiasanat: sulautetut järjestelmät, vianetsintä, mikroprosessori

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree programme in Business Information Systems
Option of Software Development

RAUMAN, MIKKO:
Troubleshooting Software Problems in Embedded Systems

Bachelor's thesis 32 pages
November 2019

The commissioner for this thesis was UnSeen Technologies Oy. UnSeen had launched a project based on embedded systems. The project required training for the employees in embedded debugging.

The objective of this thesis was to educate UnSeen's new and existing employees in embedded debugging and to research the tools and methods which would be applicable in it. This investigation was to be used in choosing debugging methods and tools for the project.

The results were obtained by researching the similarities and differences in regular software debugging and embedded software debugging. The different tools and methods were then evaluated in viability based on scenarios that happened during the project. The chosen methods were runtime debugging, test driven development and hardware instrumentation.

Runtime debugging was found to be relatively unchanged in embedded systems. Developers can still employ debugging features in their integrated development environment. Test driven development was not a viable option at the time for the lack of distinct testable entities and constraints introduced by hardware. In hardware instrumentation power usage profiling was found to be a logical enhancement to runtime debugging. Documentation for introducing the tools used in the project were created in UnSeen's internal network.

Key words: embedded systems, debugging, microprocessor

SISÄLLYS

1 JOHDANTO ... 7
1.1 Toimeksiantaja ... 7
1.2 Käsiteltävän alustan esittely ... 7
1.3 Virheenjäljityksen prosessi ... 9
2 MENETELMIEN ESITTELY ... 12
2.1 Ohjelmiston ajonaikainen testaus ... 12
2.1.1 Käyttötarkoitukset ... 12
2.1.2 Soveltuvuuden arviointi ... 13
2.2 Testivetoinen kehitys ... 13
2.2.1 Käyttötarkoitukset ... 14
2.2.2 Soveltuvuuden arviointi ... 14
2.3 Instrumenttipohjainen vianetsintä ... 14
2.3.1 Käyttötarkoitukset ... 15
2.3.2 Soveltuvuuden arviointi ... 15
3 TYÖKALUT ... 17
3.1 Ohjelmistoympäristön virheenjäljittäjä ... 17
3.1.1 Yhteys käytettyyn menetelmään ... 17
3.1.2 Havaintoja ... 18
3.2 Yksikkötestausympäristö ... 18
3.2.1 Yhteys käytettyyn menetelmään ... 18
3.2.2 Havaintoja ... 19
3.3 Virranmittausyksikkö ... 20
3.3.1 Yhteys käytettyyn menetelmään ... 20
3.3.2 Havaintoja ... 21
4 KÄYTTÖTILANTEET ... 22
4.1 Ohjelman ajon keskeytyminen ... 22
4.1.1 Esimerkkitilanne ... 22
4.1.2 Käytetyt menetelmät ja työkalut ... 23
4.1.3 Prosessi ... 24
4.2 Looginen virhe sovelluksessa ... 24
4.2.1 Esimerkkitilanne ... 25
4.2.2 Käytetyt menetelmät ja työkalut ... 25
4.2.3 Prosessi ... 25
4.3 Virhe laitteiston suunnittelussa ... 26
4.3.1 Esimerkkitilanne ... 26
4.3.2 Käytetyt menetelmät ja työkalut ... 27

4.3.3 Prosessi 27
5 POHDINTA 30
LÄHTEET 32

LYHENTEET JA TERMIT

C	Sulautetujen järjestelmien kehitykseen soveltuva ohjelmointikieli.
Debuggeri	Vianjäljityksen mahdollistava ohjelmisto tai laitteisto
FPC	Flexible Printed Circuit. Painettu piirilevy.
GPS	Global Positioning System. Satelliitteihin perustuva paikannusjärjestelmä
IoT	Internet of Things. Internetin kautta mahdollistettu laitteiden tiedonsiirto, etäseuranta ja etäohjaus.
nB-IoT	Narrowband Internet of Things. Teknologiastandardi IoT:n toteuttamiseen puhelinverkon yli.
modeemi	Modulaattori/Demodulaattori. Laite joka mahdollistaa yhteyden puhelinverkon yli.

1 JOHDANTO

1.1 Toimeksiantaja

Toimeksiantajana opinnäytetyössä on UnSeen Technologies Oy, joka lyhennetään tästä eteenpäin Unseen. Unseen on vuonna 2015 perustettu tamperelainen startup yritys. Unseen tarjoaa tuotteena asiakkaille Unseen Labs -projektia, jossa asiakkaan teknologiaongelmaan haetaan viikossa ratkaisua. Nämä ovat yleensä pienimuotoisia konsulttipalveluja, joista yleensä saadaan aikaiseksi uusia projekteja esimerkiksi prototyyppien valmistuksen muodossa. Unseen on valmistanut tuotteita aina laitetasolta asti pilvipalveluihin saakka.

Kun Unseenissa aloitettiin kehittämään omaa sulautettua järjestelmää, tarvittiin sovelluskehitystiimiin lisää työntekijöitä. Koska nykyisillä työntekijöillä ei tuolloin ollut kokemusta ohjelmistokehityksestä sulautetuille järjestelmille, nähtiin tarve tuottaa eräänlainen siirtymäohje vianetsintään liittyen nykyisiä ja tulevia työntekijöitä varten.

1.2 Käsiteltävän alustan esittely

Esineiden internet on viime vuosina noussut teollisuudessa kiinnostuksen kohteeksi. Tähän on vaikuttanut teknologian ja erityisesti elektroniikan komponenttien kehittyminen vähävirtaisempaan suuntaan. Nykyiset mikrokontrollerit ja mobiiliverkkoon yhdistävät modeemit vievät niin vähän virtaa, että näitä sisältäviä alustoja voi pitää aktiivisena pelkällä paristolla jopa useita kuukausia.

Siinä missä perinteisessä internetissä käyttäjien kesken jaetaan resursseja, jaetaan esineiden internetissä dataa. Tällaista dataa voi olla mitä vain, mikä nähdään hyödylliseksi ottaa talteen laitteelta. Yleensä esineiden internetiin liitettävin alustoihin on liitetty joko omia sensoreja, tai ne on asetettu välittämään dataa kolmannen osapuolen laitteista. Periaatteessa mitä tahansa mitä voi tulkita digitaaliseksi dataksi voidaan välittää esineiden internetin yli.

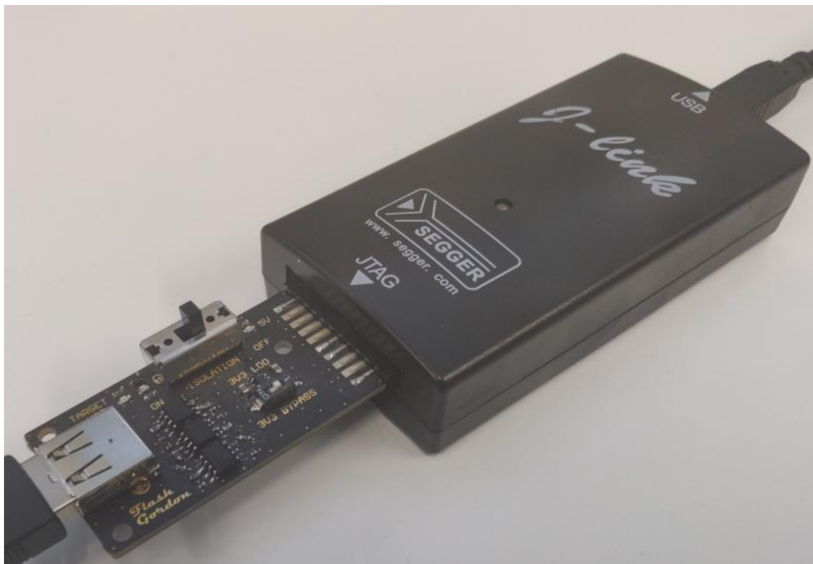
Tällaisten laitteiden verkolle on yleensä ominaista se, että sen yksittäiset laitteet ovat yleensä hajautetut laajalle alueelle. Tätä on edistänyt viime vuosina 4G-verkossa toimivien narrowband Internet of Things (nB-IoT) modeemien kehitys. Nämä mahdollistavat laitteiden sijoittamisen niin sisä- kuin ulkotiloihinkin ilman, että kuuluvuus kärsii käyttökelvottomalle tasolle.

Unseen on aloittanut ensimmäisen oman tuotteen kehittelyä. Razr-alustaan kuuluu mikrokontrolleri, nB-IoT modeemi, GPS-piiri ja joukko ympäristöä mittaavia sensoreita. Mikroprosessori on laitteen ohjelmoitava osa, jonka toimintalogiikasta sovelluskehittäjä vastaa itse. Paikantamisesta vastaava GPS-piiri käyttää hyväksi GPS ja GLONASS satelliittijärjestelmiä. Alustan kytkee internetiin modeemi, jota ohjataan mikrokontrollerin avulla selkokielisin komennoin. Kuvasta 1 näkee, että alustalla on sekä valmiiksi asennetut piirit, että mahdollisuus laajentaa alustaa siihen tarkoitetun liitännän avulla. Nämä FPC liitännät ovat tarkoitettu painettujen piirilevyjen tai -kaapelien kytkemiseen laitteeseen. Razr-alustalla Unseen tarjoaa asiakkaille dataa palveluna. Sitä varten on tuotettu verkkoselaimella käytettävä laitehallinta ja datanvälitysmahdollisuudet.

Kuva 1. Razr-alusta.

Razr-alusta tulee sen kehityskaaren aikana muuttumaan niin fyysisesti kuin ohjelmiston tarpeiden mukaan. Nopeat muutokset laitteistossa ja ohjelmistoissa vaativat, että yrityksellä on selkeät ohjeet ohjelmiston virheenjäljittämiseen niin, että jokainen Unseenin ohjelmistokehittäjä pystyy etsimään virheitä ohjelmistosta.

Järjestelmäalusta itsessään aiheutti tiettyjä rajoitteita tutkittaville työkaluille ja menetelmille. Mikroprosessorina järjestelmässä on ARM prosessoriarkkitehtuuriin perustuva Nordic Semiconductor nRF52. Virheenjäljitykseen käytettävä laite eli debuggeri on Segger Microcontroller J-Link BASE. Debuggeri tarvitsee myös erikseen valmistettavan osan laitteen liittämiseen Razr-alustaan. Tämä erillinen laite näkyy yhdessä debuggerin kanssa kuvasta 2. Näiden laitteiden asettamilla ehdoilla tutkittiin eri työkaluja ja menetelmiä virheenjäljityksessä.



Kuva 2. Segger J-Link päätelaite Razr-lisäosan kanssa.

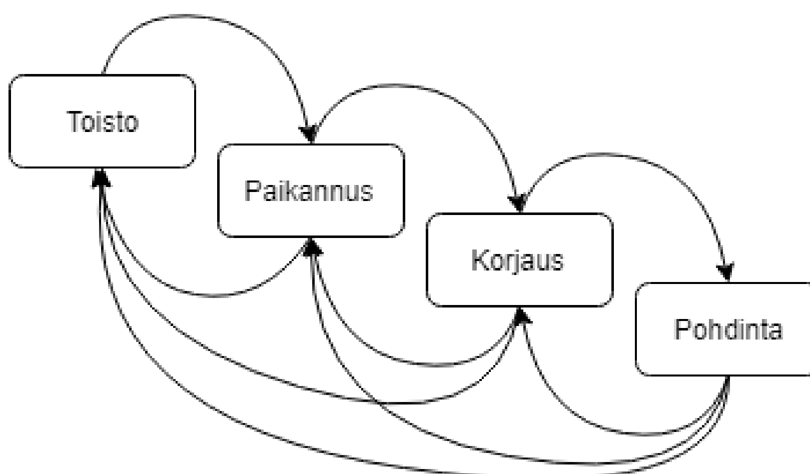
1.3 Virheenjäljityksen prosessi

Kun lähdetään tutkimaan eri työkaluja ja tapoja jäljittää virheitä kohdelaitteiston ohjelmistosta, on hyvä sisäistää mitä tarkoitetaan virheenjäljityksen prosessilla. Ohjelmiston voidaan sanoa olevan virhetilassa, kun laite ei käyttäydy odotetulla

tavalla, ei toimi ollenkaan tai pysähtyy kesken ohjelmiston ajon odottamattomasti. Virhetilan sattuessa on sovelluskehittäjän aloitettava virheenjäljitysprosessi.

Virheenjäljityksen prosessi koostuu karkeasti ottaen neljästä eri osasta: virheen toisto, paikannus, korjaus ja syyn pohdinta. Tämä prosessi ei ole yleensä täysin suoraviivainen alusta loppuun, vaan usein kesken prosessin voi joutua aloittamaan aikaisemmasta vaiheesta. (Butcher 2009, luku 1.)

Vianetsintä aloitetaan ongelman luotettavan toiston saavuttamisella. Kun ongelmaa pystyy kuvailemaan helposti, voi sen tuntomerkkien avulla etsiä tiimin ulkopuolelta apua (Spinellis 2016, luku 2). Vianjäljityksen eri vaiheiden suhde toisiinsa löyty kuvasta 3. Vianetsintä koostuukin oikeastaan kahdesta erillisestä osasta. Kun tapahtuva ongelma on saatu luotettavasti toistumaan, tulee päättää mistä kohtaa laitteistoa lähdetään poissulkemaan vaihtoehtoja ongelman synnystä. Jos sovelluskehittäjällä ei ole selvää tietoa ongelman sijainnista, on helpoin tapa käyttää puolitushaun menetelmää, jossa toimivat osat poissuljetaan testaamalla aina puolesta välistä molempiin suuntiin laitteistoa.



Kuva 3. Virheenjäljityksen prosessi (Butcher, 2009)

Vian korjaamisen vaiheeseen kuuluu myös ongelman poistumisen todentaminen. Tämä on helpointa silloin, kun sovelluskehittäjällä on jo varma tapa toistaa ongelma. Hankalan virheen paikannuksen vaiheen jälkeen tärkeintä sovelluskehittäjälle on ottaa oppia tilanteesta ja käydä jollain tasolla läpi kokemukset tilanteesta. Tämän pohdinnan vaiheen voi suorittaa itselle sopivalla

tavalla, ja parhaiten kokemuksesta oppii tekemällä pienimuotoisen raportin ja käymällä suullisesti läpi muun kehitystiimin kanssa.

2 MENETELMIEN ESITTELY

2.1 Ohjelmiston ajonaikainen testaus

Kun ohjelmistoa suoritetaan kohdelaitteistossa, voidaan ajon aikana jäljittää tapahtuvia loogisia virheitä eri tavoin. Näistä yleisintä kutsutaan ohjelmiston instrumentoinniksi. Instrumentointi on osa ohjelmiston koodia, joka ei suoraan vaikuta ohjelmiston kulkuun vaan tarjoaa käsityksen, miten ohjelmisto toimii. Mahdollisesti vanhin virheenjäljitysmenetelmä on väliaikaisen tulosteiden lisääminen koodiin. (Butcher 2009, luku 3.)

Toinen tapa virheenjäljittämiseksi on ohjelmiston koodin ajonaikainen läpikäyminen rivi riviltä. Virheenjäljitysohjelman avulla annetaan pysähtymiskäsky laitteelle tiettyyn kohtaan ohjelmiston suoritusta. Ohjelmiston pysähtyessä virheenjäljetysohjelma hakee mikroprosessorin muistista sen hetkiset arvot. Näitä arvoja pystytään seuramaan samalla kun prosessorille annetaan käsky siirtyä ohjelman suorituksessa seuraavalle riville.

Joskus on hyödyllistä myös ohjelmiston suorituksen pysäytyksen aikainen koko muistin tilanteen tallentaminen kehityskoneelle, jotta esimerkiksi mahdolliset ajonaikaiset muistivuodot voidaan havaita.

2.1.1 Käyttötarkoitukset

Ohjelmiston instrumentoinnilla saavutetaan yleensä minimissään se, että kehittäjä pystyy seuraamaan ohjelman kulkua reaaliaikaisesti tulosteiden avulla. Näin pystytään esimerkiksi saamaan tulosteita ulkoisilta sensoreilta ja varmistamaan, että niiden antamat arvot vastaavat odotettuja. Sulautetuissa järjestelmissä tulosteiden seuranta antaa yleensä ensimmäisen osoituksen ohjelmistossa tai laitteistossa ilmenevästä viasta.

Yllättävien kaatumisten paikantamiseksi pysäytyspisteet ovat yleensä ensimmäinen ja paras tapa saada kiinni virhetilanteesta. Jos pysäytyspisteiden

käyttö ei ole mahdollista, käännöksen aikana luotu vedostiedosto helpottaa vianetsintää silloin kun käytössä on pelkästään ohjelman ajon aikana saadut tulosteet.

2.1.2 Soveltuvuuden arviointi

Instrumentointi voidaan jakaa kahteen eri luokkaan. Ohjelmiston kulkuun vaikuttavaan instrumentointiin kuten tulosteiden antoon tai valon vilkuttamiseen piirilevyllä, ja ohjelmiston kulkuun vaikuttamattomaan instrumentointiin, jossa debuggerin päätelaite kuuntelee suorittimen komentoja lennosta.

Tulosteiden lisääminen ohjelmistoon ei ole ainoastaan hyvä asia. Sulautetuissa järjestelmissä ohjelmiston suorittaminen on aikaherkkää, ja tulosteiden kirjoittamiseen käytetty aika on pois itse hyötykoodin ajolta. Instrumentointi voi itsessään aiheuttaa yllättäviä ongelmia, kun kehitysympäristöstä siirrytään tuotantoympäristöön, josta instrumentointi on poistettu. Kehittäjän onkin tärkeää pitää huoli siitä, että ohjelmiston toimivuus testataan tarpeeksi usein myös tuotantoympäristössä kehityksen aikana.

2.2 Testivetoinen kehitys

Testivetoinen kehitys on ennaltaehkäisevää virheenjäljittämistä, joka perustuu automaattisiin yksikkötesteihin. Kun yksikkötestaus voidaan automatisoida, pystytään ohjelmiston tuotanto toteuttamaan testivetoisesti. Jokaisessa uudessa tuotannon vaiheessa tavoitteena ennen koodin kirjoittamista on luoda yksikkötestit, joilla mallinnetaan haluttu toiminnallisuus. Näiden testien on tarkoitus epäonnistua aluksi. Testien kirjoittamisen jälkeen vasta luodaan koodissa toteutus, joka läpäisee testit. (Grenning 2011, sivu 4.)

Testivetoisessa kehityksessä toteutus luodaan hyvin lyhyissä väleissä niin, että haluttu toiminnallisuus testataan jo ennen jokaista muutosta. Näin haluttu toiminnallisuus voidaan testata kohdelaitteistossa ja sen ulkopuolella ennen kuin

se tuodaan osaksi ohjelmistoa. Kehityslaitteistolla onkin mahdollista ajaa enemmän testejä nopeammin kuin kohdelaitteistossa.

2.2.1 Käyttötarkoitukset

Testivetoisen kehityksen pääasiallinen tarkoitus on estää loogisten virheiden pääsy ohjelmistoon. Ohjelmistoon tarkoitetuille funktioille annetaan jonkinlainen syöte, ja funktion pitäisi antaa ennalta-arvattavia arvoja takaisin. Näin funktiolle voidaan määritellä hyväksyttävät syötteet ja tulosteet, ja funktio voidaan testata kohtuullisilla raja-arvoilla. Kun testit on saatu läpäisemään, tai odotetusti hylkäämään, voidaan testattu koodi melko varauksetta viedä sellaisenaan tuotantoympäristöön.

2.2.2 Soveltuvuuden arviointi

Testivetoinen kehitys on perinteisessä korkeamman tason ohjelmistokehityksessä erittäin hyödyllinen tapa välttää virheitä. Sulautettujen järjestelmien ohjelmiston kehityksessä on kuitenkin sellaisia ominaispiirteitä, jotka vaikeuttavat testivetoisen kehityksen toteuttamista.

Yksikkötestauksessa on tapana mallintaa ulkoiset laitteet. Niistä luodaan jäljitelmiä, jotka toimivat kuten oikeassa tilanteessakin, mutta kuitenkin aina ennalta-arvattavasti. Näin ulkoisten laitteiden omat virhetilat eivät pääse vaikuttamaan testien lopputulokseen. Sulautetuissa järjestelmissä voi helposti joutua tilanteeseen, jossa jäljitelmiä joutuu tekemään enemmän kuin varsinaisia testejä. Varsinkin kun uutta alustaa ollaan ottamassa käyttöön, täytyy jokainen ulkoinen laite, reaaliaikakäyttöjärjestelmä ja kommunikaatiokerroksen ajurit jäljitellä.

2.3 Instrumenttipohjainen vianetsintä

Sulautetuissa järjestelmissä hyvin usein myös laitteisto on kehitysasteella yhtä aikaa ohjelmiston kanssa. Kehittäjä voi joutua tilanteeseen, jossa omasta mielestä ohjelmisto on loogisesti virheetön, mutta laitteisto ei silti toimi halutulla tavalla. Hyvin usein tällaiset ongelmat tapahtuvat ohjelmoitavan mikroprosessorin ulkopuolella, jolloin niihin on erittäin haastava päästä kiinni.

2.3.1 Käyttötarkoitukset

Yksinkertaisimmillaan vikaa voidaan hakea pelkällä yleismittarilla, jolloin mikroprosessorilta voidaan testata sähköinen yhtäjaksoisuus oletetuilta pinneiltä haluttuun päämäärään. Näin voidaan helpoimmillaan tarkastaa, onko ”johto kiinni”. Vaikeimmillaan laitteistoa voidaan instrumentoida oskilloskoopilla, mutta erittäin harvoin ohjelmistokehittäjän odotetaan hallitsevan kyseistä laitetta. Ohjelmistokehittäjän tarpeisiin soveliain työkalu onkin virrankulutusta profiloiva laite, joka tuottaa dataa virrankulutuksesta ajan funktiona. Näin virrankulutus voidaan sitoa ohjelmiston tuottamiin tulosteisiin ja sitä kautta löytää virhetilanteen mahdolliset aiheuttajat.

Ylimääräinen virrankulutus sulautetuissa laitteissa voidaan myös laskea virhetilanteeksi. Erityisesti silloin kun laitteen on tarkoitus toimia akku- tai paristovirralla, on haluttavaa karsia tarpeeton virrankulutus laitteistossa. Käyttövirran profilointi tuottaa myös yleisellä tasolla tietoa laitteiston toiminnasta eikä pelkästään mikroprosessorin toiminnasta. Sovelluskehittäjien on kuitenkin hyvä muistaa, että ennen kuin virrankulutusta aloitetaan optimoimaan, on hyvä perustaa lähtötaso sovellukselle jota optimoidaan (Oshana 2013, luku 13).

2.3.2 Soveltuvuuden arviointi

Hyvin usein sulautetun järjestelmän sovelluskehittäjän viimeinen vaihtoehto on epäillä vikaa laitteistossa itsessään. Laitteiston osat toimivat erikseen, mutta voivat sähköisellä tasolla vaikuttaa myös toisiinsa. Joskus nopea testaus yleismittarilla voi auttaa helposti, ja joskus yhtämittainen virtamittauksen profilointi tuo ymmärystä laitteen käyttäytymiseen.

Vaikka ensituntumalta voi tuntua oudolta pitää ohjelmiston ulkopuolisia tekijöitä ohjelmistokehityksen ongelmana, todellisuudessa laitteiston heikkouksia voi hyvin usein kompensoida ohjelmistotasolla. Koska perinteisessä ohjelmistokehityksessä ei todennäköisesti tule helposti epäiltyä vikaa laitteistossa, on hyvä virhetilanteen sattuessa testata ohjelmiston kulku useammalla eri laitteella.

3 TYÖKALUT

3.1 Ohjelmistoympäristön virheenjäljittäjä

Razr-alustaan on luonnollisesti valikoitunut vapaaseen lähdekoodiin perustuva GNU-projektin kääntäjien kokoelma. Siitä on johdettu ARM-prosessoreiden kohteisiin tarkoitettu työkalukokoelma. Tähän luetaan mukaan myös gdb eli GNU Debugger, joka on GNU-projektin virheenjäljitysohjelma.

Gdb on komentoriviltä ajettava sovellus, jota voi käyttää myös verkon yli. Monet graafiset kehitysympäristöt tukevatkin GDB:n käyttöä ainakin jollain tasolla suoraan näiden omista sovelluksista. Kun kehitysympäristö on konfiguroitu etukäteen oikein, on virheenjäljityksen aloittaminen erittäin suoraviivaista suoraan kehitysympäristön sovelluksesta.

3.1.1 Yhteys käytettyyn menetelmään

Debuggeri mahdollistaa pysäytyskäskyjen asettamisen haluttuun kohtaan koodissa, josta voidaan aloittaa ohjelmiston kulun seuranta riveittäin. Yleensä tämä on hyödyllistä siten, että sovelluskehittäjä pystyy seuraamaan viittauksien oikeellisuutta ja muistin eheyttä. Tätä tapaa voi kuitenkin käyttää tehokkaasti vain silloin kun mikroprosessorin oma Bluetooth-pino on pois käytöstä. Sen käyttö estää virheenjäljitystilanteessa prosessorin vapaan suorituksen jatkamisen, aiheuttaen yleensä keskeytyksen ja ohjelmiston kaatumisen. (Veilleux, 2017.) Näin ollen on ajan säästämiseksi hyvä asettaa pysähtymiskäsky mahdollisimman lähelle arvioitua vikatilaa.

Toinen haaste on alustan ohjelmistossa käytettävä reaaliaikakäyttöjärjestelmä. Sen tehtävänä ohjelmistossa on ajastaa tehtäviä ja jakaa niille tasaisesti prosessorilta ajoaikaa. Kun laitteiston oma kello toimii itsenäisesti pysähtymiskäskystä huolimatta, menee reaaliaikakäyttöjärjestelmä yleensä sekaisin sen yrittäessä vaihtaa tehtävää virheenjäljityksen ollessa päällä.

3.1.2 Havaintoja

Mikroprosessorin valmistaja on jo ratkaissut nämä ongelmat antamalla keskeytyspyyntöjen toimia samalla kun virheenjäljityssessio on käynnissä. Tällaista virheenjäljitystilaa kutsutaan nimellä Monitor Mode Debugging. (Veilleux, 2017.) Tämän tilan käyttöönotto lisää huomattavasti virheenjäljityksen mahdollisuuksia, mutta toistaiseksi se vaatii käytettävästä debuggerin päätelaitteesta kehittyneemmän version.

Ohjelmiston kulkua on mahdollista seurata myös ilman ohjelman kulkuun puuttumista yksittäisten käskyjen tasolla, jos debuggerin päätelaite tukee tätä toimintoa. Tämä vaatii myös kohdeprosessorilta tarvittaviin pinneihin fyysiset yhteydet. Tästä tavasta on eritoten hyötyä erittäin haastavien vikatilanteiden paikantamisessa, jossa ohjelmisto yllättäen kaatuu.

3.2 Yksikkötestausympäristö

Sulautetuille järjestelmille ja erityisesti C kielelle tarkoitettu testiympäristö Unity on avoimeen lähdekoodiin perustuva testiympäristö. Sen etuna ovat suhteellisen pieni muistijalanjälki ja laajennettavuus eri lisäosin. Moduulilla nimeltä CMock voidaan luoda jäljitelmiä ulkoisista laitteista ja Cexception-moduulilla voidaan simuloida poikkeuksia. Näiden avulla päästään melko lähelle perinteisessä ohjelmistosuunnittelussa käytettäviä menetelmiä.

On myös mahdollista kirjoittaa testiohjelmia suoraan kohdelaitteistoon. Tämä on hyödyllistä esimerkiksi silloin kun halutaan testata miten mikroprosessorista ulkoinen laite käyttäytyy eri tilanteissa. Nämä ohjelmat ovat yleensä yksikkötestauksesta erillisiä, mutta silti erittäin hyödyllisiä laitetason integraatiossa.

3.2.1 Yhteys käytettyyn menetelmään

Unity testiympäristön käyttö on melko suoraviivaista, koska se on hyvin suoraan rinnastettavissa perinteisessä ohjelmistokehityksessä käytettävään testivetoiseen kehitykseen. Termistö ja ajatusmaailma sulautettuihin järjestelmiin siirryttäessä pysyykin melko muuttumattomana.

Joitakin eroja kuitenkin on. Esimerkiksi lopullista kohdelaitteistoa ei välttämättä ole saatavilla kaikille kehittäjille yhtä aikaa. Kohdelaitteiston mikroprosessorille on kuitenkin usein saatavilla valmistajan kokeilu- tai kehityspiirilevyjä, joilla voidaan ajaa testejä tarvittaessa. Eri alustojen käyttö vaatii ohjelmistolta siirrettävyyttä, mutta sulautetuissa järjestelmissä kohdealustan eläessä on hyvä keskittyä siirrettävyyteen kehityksen alusta asti.

3.2.2 Havaintoja

Kun ohjelmiston vaatiman muistin määrä kohoaa lähelle kohdelaitteistossa olevaa muistin määrää, ei kaikkia testejä voi ajaa kohdelaitteistossa kerralla. Silloin ajettavat testit pitää muuttaa pienempiin käännösosiin, jotka ajetaan vuorotellen kohteessa. Tämä kuitenkin tuo testausprosessiin omat ongelmansa. Haihtumattomien muistien fyysinen tyhjennys ja uudelleenkirjoitus kuluttaa muistia (Richter, D. luku 4). Näin ollen pienien testausohjelmien peräkkäinen ajo kohdelaitteistossa ei välttämättä ole kovin mielekästä. Mikroprosessorin haihtumattomaan muistiin voi kirjoittaa vain rajallisen määrän kertoja, ja toistuva testiohjelmien ajo kuluttaa muistin elinaikaa huomattavan nopeasti. Razr-alustassa käytettävän mikroprosessorin haihtumaton muisti on luokiteltu vähintään kymmentätuhatta kirjoituskertaa varten (nRF52840 Product Speficifation. 2018). Tämä tarkoittaa, että noin kymmenen tuhannen kirjoituskerran jälkeen laitteen valmistaja ei voi taata, että haihtumattomassa muistissa on sama data, mitä sinne on kirjoitettu.

Muistin kulumiseen testauksessa voi tulevaisuudessa auttaa kohdelaitteiston simulointi, jossa mikroprosessoria jäljitellään isäntälaitteistossa sovellustasolla. Kohdelaitteiston simulointi voisi tuoda hyvän määrän erilaisia hyötyjä. Simulaattorilla voi testata asioita, joita ei normaalilla laitteistolla pysty. Tällaisia asioita ovat esimerkiksi kirjoitussuojattuun muistin muuttaminen, ajon

keskeyttäminen ja palauttaminen ilman pelkoa synkronoinnin häiriintymisestä. Simulaattori vapauttaa sovelluskehittäjän myös laitteiston saatavuudesta johtuvista ongelmista. (Ingianni, 2018.)

Laitteiston simuloinnin hyödyt ovat hyvin selvät, mutta toistaiseksi Razr-alustassa käytettävään mikroprosessoriin ei ole virallista simulaattoria laitteen kehittäjän toimesta. Markkinoilta löytyy kyllä yksi kehitysvaiheessa oleva maksullinen palvelu, joilla ilmeisesti pystyy Razr-alustasta löytyvää mikroprosessoria simuloimaan, mutta palvelu on sen verran uusi ja siitä ei ole juuri tietoa netissä. Näin ollen sitä on erittäin hankala käsitellä varteenotettavana vaihtoehtona, ainakin kunnes käytetyn mikroprosessorin valmistaja on antanut palvelulle siunauksen.

3.3 Virranmittausyksikkö

Virrankulutuksen profilointiin kuuluu yleensä siihen tarkoitettu laite, ja laitetta käyttävä ohjelmisto. Razr-alustaa kehitettäessä vaihtoehtoja on ollut kaksi. Jos ohjelmiston virrankulutuksen profilointiin tarvitaan vain pelkän mikroprosessorin tutkimista, voidaan käyttää Nordic Semiconductorin omiin kehityslaitteisiin tarkoitettua Power Profiler Kit -lisäosaa.

Razr-alustan virrankulutusksen profiloimiseen valikoitui Otii Arc -niminen laite. Tällä ulkoisella laitteella voidaan mitata koko alustan virrankulutusta ja asettaa sähkövirran heilahtelulle turvalliset raja-arvot, joiden ylittyessä laite katkaisee virrat. Tämä on hyvä turva esimerkiksi vahingollisten oikosulkujen aiheuttamien ongelmien estämiseen laitteistossa.

3.3.1 Yhteys käytettyyn menetelmään

Kun virrankulutuksen graafi yhdistetään kohdelaitteistosta saataviin tulosteisiin, voidaan yksinkertaisen hahmontunnistuksen avulla paikantaa laitteistossa tapahtuvia toimintoja, joita ei perinteisten tulosteiden avulla saada näkyviin.

Sähkötekniikasta oppimattomalle sovelluskehittäjälle menetelmän käyttäminen tarkoittaa pääsääntöisesti

Alustassa käytettävä mikroprosessori pohjautuu tapahtumapohjaiseen ohjelman ajoon ja kun oheislaitteelle annetaan suorituskäsky, sitä ei pysty ohjelmistossa seuraamaan muuten kuin tuloksia odottelemalla. Samoin kun ulkoisia piirejä kuten modeemia tai GPS-piiriä komennetaan mikroprosessorilta, eivät nämä tuota perinteisessä mielessä tulosteita. Ne kuitenkin tuottavat virrankulutuksessa helposti tunnistevia kuvioita, jotka kertovat kehittäjälle totuuden siitä, mitä laitteistossa oikeasti tapahtuu.

3.3.2 Havaintoja

Vaikka virrankulutuksen profilointi on tehokas väline sulautettujen järjestelmien vianetsinnässä, Ilmaisee tapa yleensä vain virheen olemassaolosta. Sovelluskehittäjän vastuulle jääkin usein määritellä virheen alkuperä.

Virrankulutuksen profilointi on myös vaivaton työkalu käyttää yhdessä ajonaikaisen kehityksen kanssa, ja se tuo lisäarvoa kehitykseen, jos sitä on mahdollisuutta käyttää jatkuvasti. Tämä ei aina kuitenkaan ole mahdollista jos kehitystiimissä on rajallinen määrä laitteita.

4 KÄYTTÖTILANTEET

Käyttötilanteet, joissa valittuja laitteita ja menetelmiä on päästy oikeasti hyödyntämään ovat olleet arvokkaita kokemuksia. Kun tilanne vaatii menetelmän tai työkalun vaihtoa, kehittäjän hyvä tietää merkit koska nykyinen menetelmä on tehoton tai turha. Näiden merkkien tunnistamiseen harjaantuu parhaiten kokemuksella, mutta yleensä tilanteet on hyvä analysoida tulevaisuutta varten.

Tässä kappalessa esitellyt tilanteet ovat Razr-alustan kehityksessä tapahtuneita tilanteita. Näissä tilanteissa eri työkalujen merkitys korostui huomattavasti muihin nähden, ja niiden oma arvo korostui. Näiden tilanteiden perusteella on myös arviot menetelmien ja tyokalujen soveltuvuudesta tehty.

4.1 Ohjelman ajon keskeytyminen

Yleisin ja tutuin tilanne sovelluskehittäjille C kielessä on virheellisten tai tyhjien osoittimien aiheuttama sovelluksen ajon keskeytyminen. Sovellus, jonka ajo keskeytyy joka kerta samassa kohtaa on yleensä suoraviivainen korjata.

Perinteisen sovelluskehityksen menetelmät purevat usein keskeytyvän sovelluksen virheen paikantamiseen. Oman hankaluuden voivat kuitenkin tuoda esimerkiksi laitteiston valmistajan omat suljettuun lähdekoodiin perustuvat sovelluksen osat, joita ei selkokielisenä voi selata.

4.1.1 Esimerkkitilanne

Ohjelmaan totetutettiin yksinkertainen viestijono, jossa modeemin vastaanottamat viestit otettiin tekstimuodossa vastaan, muunnettiin dataksi ja lopulta lähetettiin pääteohjelmalle käsiteltäväksi. Kun viesti muutettiin datamuotoon, muistista varattiin tilaa datalle. Kun viesti oli välitetty pääteohjelmalle, muistista varattu tila vapautettiin. Kun ohjelman annettiin ajaa pitemmän aikaa, ohjelmiston ajo keskeytyi kaatumiseen.

Ohjelmassa olevan muistinvapautuksen funktion listaus on kuvassa 4. Tässä listauksessa on listattu vain virhetilan tapahtumiseen olennainen osa koodista, vaikka sitä ei ongelman syntyessä tiedetykään. Listaus on myös funktionaalisesti identtinen oikeassa tilanteessa sattuneen koodin kanssa, mutta sitä on siistitty luettavuutta varten.

```
11  void receive_data()
12  {
13      uint8_t* data;
14      size_t data_len;
15
16      while (true)
17      {
18          data = NULL;
19          data_len = 0;
20
21          xQueueReceive(data_queue, &data, portMAX_DELAY);
22
23          data_len = get_buf_len(data);
24
25          if (data_len > 0)
26          {
27              handle_data(data, data_len);
28          }
29
30          nrf_free(data);
31      }
32  }
```

Kuva 4. Esimerkkitilanteen koodilistaus

4.1.2 Käytetyt menetelmät ja työkalut

Kun ohjelman virheenjäljityksessä käytetään gdb ohjelmaa, osaa laitteistossa oleva virhetilan tunnistus pysäyttää debuggerin samalla tavalla kuin käsky olisi annettu kehityskoneesta. Tämä antaa mahdollisuuden tutkia, mitä mikroprosessorin rekistereissä on, ja sieltä löytyy myös viimeisimmän kutsutun funktion osoite. Tämän tiedon avulla voidaan ohjelman ajo pysäyttää tämän funktion alkuun ja siitä käydä ohjelmiston kulku rivi riviltä läpi, kunnes virhe tapahtuu.

Laitteen ohjelmiston virheenkäsittelijä näyttää myös mikroprosessorin rekistereiden tilan tulosteena. Prosessorin paluurekisterin arvo näyttää myös viimeisen kutsutun funktion osoitteen muistissa. Ohjelmistoa kääntäessä on mahdollista tuottaa vedostiedoston ohjelmiston koodista. Tästä tiedostosta näkee kaikkien ohjelmassa olevien koodirivien paikat laitteen muistissa. Näitä kahta tietoa verrattaessa voidaan paikallistaa missä vika tapahtuu.

4.1.3 Prosessi

Ohjelmaa ajettaessa se kaatui aina satunnaisen ajan jälkeen listauksen riville 30 nrf_free funktion kutsuun. Tälle riville asetettaessa pysähtymiskäsky saatiin näkymään sen hetkiset muuttujat. Muuttujista kävi ilmi, että saatu osoitin dataan oli jo vapautettu muualla, mitä ilmeisimmin virheen sattuessa. Funktio get_buf_len palautti näissä tapauksissa arvon 0, jolloin listatun funktion ei olisi pitänyt yrittää vapauttaa osoittimen osoittamaa muistipaikkaa.

Korjaus tähän vikaan oli yksinkertaisesti siirtää nrf_free funktion kutsu edellisen lohkon sisään. Tästä seurasi myös, että tiimin kanssa kävimme läpi tämän datan elinkaaren ja selkiytimme sitä niin, että vastaava ei enää voisi tapahtua.

4.2 Looginen virhe sovelluksessa

Loogiset virheet eivät joka kerta tarkoita sovelluskehittäjän tekemän ajatteluvirheen aihuttemaa vikaa sovelluksessa. Esimerkiksi käytetty kohdelaitteisto tai -ympäristö voivat käyttäytyä eri tavalla kuin on odotettu. Hyvin usein laitteiston aiheuttamat loogiset virheet tapahtuvat liukulukujen käsittelyssä, joka riippuu pitkälti käytetystä prosessoriarkkitehtuurista.

Mikään ei tosin poista mahdollisuutta sovelluskehittäjän aiheuttamasta loogisesta virheestä sovelluksessa. Tällaiset virheet voivat olla erittäin hankalia toistaa, koska yleensä ne tarvitsevat erittäin tietyt olosuhteet virheen tapahtumiselle. Jos virhettä ei voi helposti toistaa, sen korjaamisen voi mennä hyvinkin paljon aikaa.

4.2.1 Esimerkkitilanne

Razr-alustassa käytettävä sovelluskerroksen pääteohjelma tarvitsee monotonisen ajan laitteen käynnistyksestä asti. Tämä on hieman hankala toteuttaa sulautetuissa järjestelmissä. Järjestelmässä käytettävä reaaliaikakäyttöjärjestelmä pitää kuitenkin sisäistä aikaa keskeytyksellä, joka tapahtuu 1024 kertaa sekunnissa. Joka kerta kun keskeytys tapahtuu, laitteen muistissa olevaa arvoa kasvatetaan yhdellä. Tätä arvoa seuraamalla voidaan laskea aika laitteen käynnistyksestä.

Pääteohjelma halusi monotonisen ajan kokonaisina sekunteina ja jäljelle jäävän osuuden nanosekuntteina. Kun ohjelmaa ajettiin pidempään, alkoi pääteohjelma käyttäytyä oudosti, ja tuntui kuin aika ei päivittyisi ohjelmalle ollenkaan.

4.2.2 Käytetyt menetelmät ja työkalut

Tämänkaltainen vika paikantuisi erittäin helposti, jos käytössä sen tapahtumisaikana olisi ollut laajamittainen yksikkötestaus ja testivetoinen kehitys. Ongelmatilanne täyttää kaikki tuntomerkit: funktion palauttamat arvot oli dokumentoitu etukäteen ja kaikki siinä käytettävät arvot olivat tiedossa. Tämän funktion olisi voinut luoda yksikkötestauksen avulla.

Tätä tapausta varten kuitenkin kirjoitettiin oma testiohjelma, jonka avulla funktiota testattiin. Koska funktio kirjoitettiin pääteohjelmaa varten, sen implementaatiota varten oli hyvä dokumentointi. Näin osattiin määritellä mitä arvoja funktion pitäisi palauttaa.

4.2.3 Prosessi

Kun vika oltiin tulosteiden avulla saatu paikannettua tähän aikaa käsittelevään funktioon piti selvittää miksi se aiheutti virhetilan. Kun testiohjelmaa ajettiin kohdelaitteistossa kaikilla 32 bittisillä luvuilla, alkoi funktio antaa virheellisiä

arvoja sekunnin murto-osille. Tämä luku piti esittää nanosekunteina, eli sen arvon piti olla jotain väliltä nolla ja yksi miljardi.

Tarkempi tutkiminen näytti, että kun sekunnin murto-osien esitystapaa keskeytyspyyntöjen laskusta yritti muuttaa nanosekunneiksi, tuli 32 bittisten lukujen katto vastaan ja tietoa katosi laskutoimituksen aikana. Logiikassa ei siis itsessään ollut mitään vikaa. Matemaattinen laskutoimitus piti jakaa useampaan osaan niin, ettei 32 bittistä esitystapaa ylitetty kummaltakaan puolelta.

4.3 Virhe laitteiston suunnittelussa

Razr-alusta on rakennettu talon sisällä alusta alkaen, ja sen suunnittelu on elänyt paljon sen kehityksen aikana. Sen takia helposti myös laitteiston kehityksessä tai sen dokumentoinnissa on voinut syntyä vikoja, joille sovelluskehittäjä ei voi itse mitään. Käyttötarkoitusten nopea muuttuminen tarkoitta myös nopeita muutoksia laitteiston käyttöön sen eri versioiden välillä, ja onkin helppo nähdä, miten vikoja voi syntyä.

Sovelluskehittäjän on suhteellisen helppo syyttää laitteistoa aiheutuvista vioista, ja jos vika ei ole ilmiselvä, on todistustaakka sovelluskehittäjällä. Tällöin on uskottavasti näytettävä, että vika tapahtuu laitteistossa. Laitteistossa olevan vian todentaminen tarkoittaa yleensä muiden virheiden mahdollisuuden eliminointia, mutta harvoissa tapauksissa myös tulosteilla ja mikrokontrollerin omilla ominaisuuksilla voidaan näyttää laitteistovika toteen.

4.3.1 Esimerkkitilanne

Razr-alustaan kiinnitettiin lisälevy, jonka avulla saatiin ulkoisesta laitteesta dataa lähetettäväksi edelleen. Lisälevy sai virtansa samasta lähteestä kuin Razr-alusta. Ohjelman ajo kuitenkin keskeytyi ja alkoi alusta jonkin aikaa sovelluksen ollessa päällä. Vian paikantamista vaikeutti se, että vika toistui satunnaisin väliajoin, jolloin vian sijoittaminen tiettyyn kohtaan ajoa oli erittäin vaikeaa.

Kun laitteistoon lisätään fyysisiä osia, on helppo olettaa ongelmien johtuvan niistä. Haastavaa on kuitenkin näyttää olettamusta toteen, jolloin on saatava jotain näyttöä siitä, että vika on laitteiston yhdistelmässä.

4.3.2 Käytetyt menetelmät ja työkalut

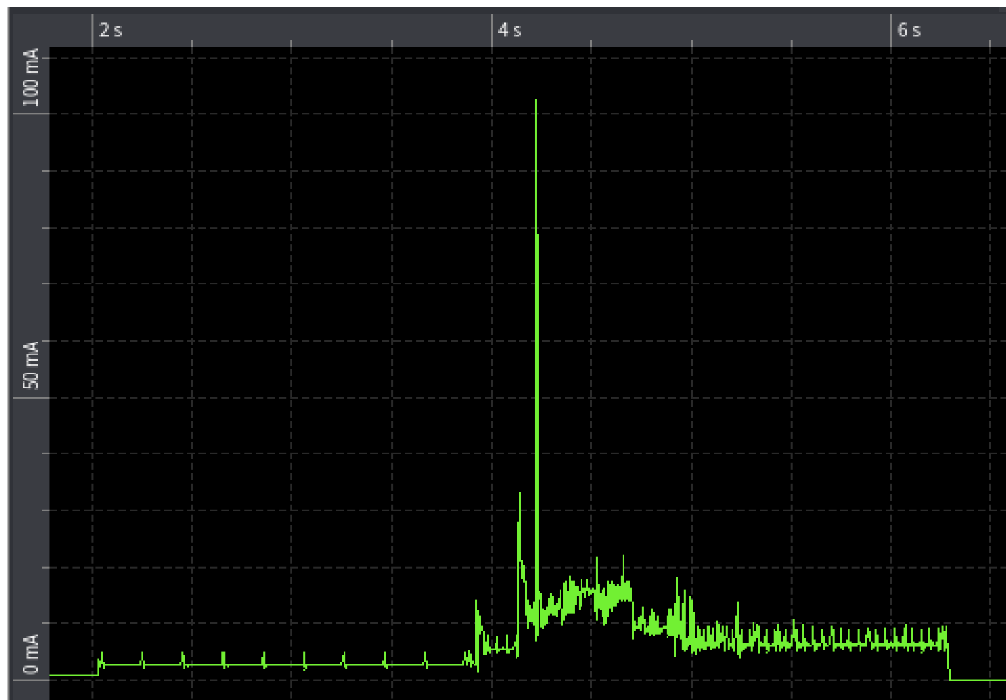
Laitteistossa esiintyvät ongelmat esiintyvät yleensä kun alustaan lisätään uusia komponentteja tai liitännäisiä. Tällaisissa tapauksissa ongelman ensimmäiset oireet tulevat ilmi instrumentoinnin avulla. Tulosteista käy nopeasti ilmi jos oheislaite ei vastaa ollenkaan. Tässä tapauksessa tosin oheislaite toimi odotetulla tavalla, ja siitä siitä saatiin haluttu data. Ensimmäinen oire ongelmasta olikin laitteen yllättävä uudelleenkäynnistyminen, jonka syytä lähdettiin etsimään tulosteista.

Itse ongelman paikantamiseksi otettiin käyttöön virranprofiloinnin lisälaite. Koska prosessorin reset aiheuttaa melko tunnistettavan käyrän korkean virtapiikin ympärille, yhdessä tulosteiden kanssa pystyi helposti toteamaan mitä muuta resetoinnin lisäksi laitteessa tapahtui. Kun pystyttiin sanomaan, että samat olosuhteet aiheuttivat resetoinnin joka kerta, voitiin ratkaisua lähteä hakemaan sovelluksen koodin ulkopuolelta.

4.3.3 Prosessi

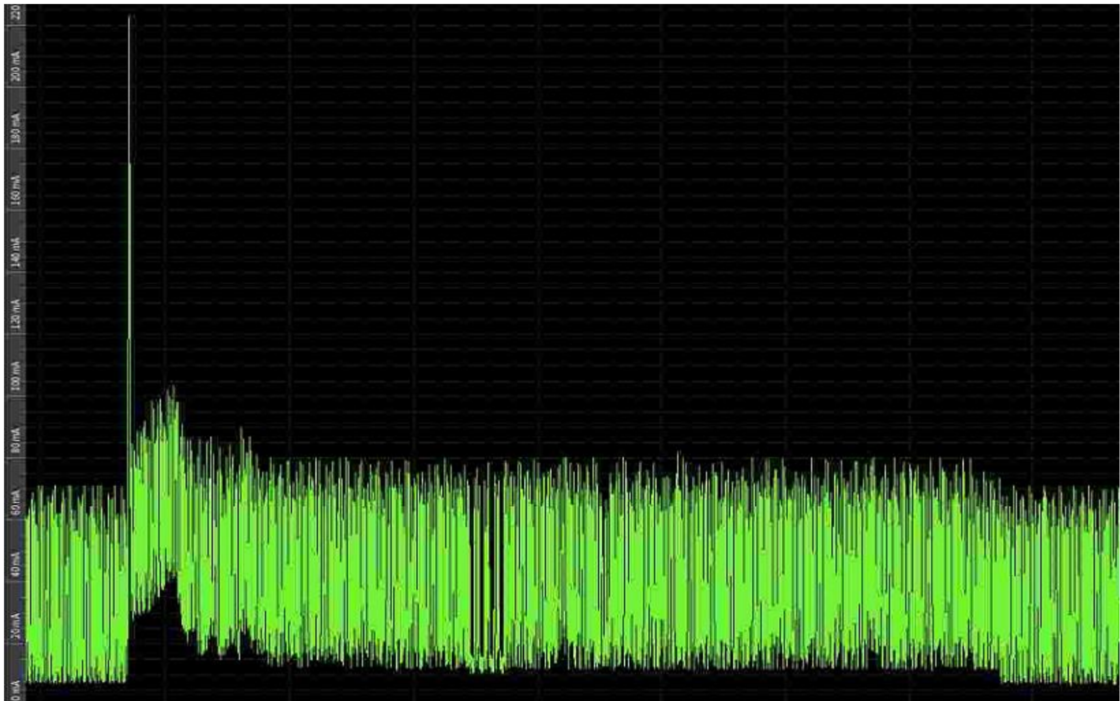
Kun mikroprosessori ajon aikana käynnistyy uudelleen, kirjoittaa se sisäänrakennettuun rekisteriin tiedon resetin lähteestä (nRF52840 Product Specification. 2018. Sivu 72). Tämän rekisterin arvo luetaan tulosteisiin aina laitteen käynnistyessä. Tässä tapauksessa rekisterin arvo näytti nollaa. Nolla tuossa rekisterissä tarkoittaa, että laite on herännyt virrattomasta tilasta tai prosessorin käyttöjännite on tipahtanut alle tietyn tason, jolloin mikroprosessori sulkee itse toimintansa suojellakseen laitetta. Koska ohjelmaa ajaessa reset rekisterin tila ei muuttunut vian sattuessa, oli vian oltava laitteistossa.

Oli enää selvitettävä mistä vian esiintymisen satunnaisuus johtui. Kuvasta 5 näkee, miltä mikroprosessorin virrankulutus näyttää normaalissa käynnistystilanteessa. Jälkimmäinen puolisko noin neljän ja viiden sekunnin välissä kaaviota näyttää sen käyrän mikä pysyy melko lailla muuttumattomana joka kerta kun laitteelle kytketään virta.



Kuva 5. Virrankulutus normaalissa käynnistyksessä

Tarkoitus oli löytää tuo sama tunnistettava käyrä kun virhe tapahtuu laitteistossa. Haastetta tähän toi se, että virrankulutuksen graafissa näkyy kaikkien laitteen osien virrankäyttö summana, jolloin tiettyjen muotojen hakeminen saattaa olla hieman haasteellista. Kuvasta 6 näkee miltä sama käynnistyminen näyttää, kun prosessori resetoituu modeemin ollessa käynnissä.



Kuva 6. Virrankulutus virhetilassa.

Vika johtui siis lopulta siitä, että kun prosessori ja modeemi olivat jo aktiivisina, kun lisälevyyn kytkettiin virta, tippui jännite prosessorilta riittävän alas virhetilan syntymiseksi. Tämä todettiin saamalla samannäköinen virrankulutuksen funktion kuvaaaja joka virhetilanteessa. Tässä kohtaa oli tarpeeksi näyttöä laiteviasta ja lyhyen palaverin jälkeen laitteistokehittäjä lisäsi levyyn kondensaattoreita tasaamaan virtapiikkiä, jolloin lisälevy toimi kuten pitikin.

5 POHDINTA

Tavoitteena oli tutkia menetelmiä ja työkaluja, mitä voitaisiin käyttää Razr-alustan kehittämiseen ja siihen kuuluvaan vianetsintään. Pienin varauksin voi sanoa, että sovelluskehittäjille tutut työkalut toimivat myös sulautetussa ympäristössä. Jos oman sovellukehitysympäristön on saanut toimimaan oikein laitteiston kanssa, ei eroa välttämättä normaaliin sovelluskehitykseen huomaa. Näin ollen isoin haaste on oikeiden asetusten saaminen sovelluskehitysympäristöön. Tosin tietyt erityistapaukset tuovat lisäehtoja sujuvaan ajonaikaiseen vianetsintään, kuten ohjelmiston riippuvaisuudet laitteiston muista osista.

Testivetoinen kehitys on hyvin suosittua nykyaikana ja se on myös onnistuneesti tuotu sulautettuihin järjestelmiin. Se ei kuitenkaan sopinut ainakaan opinnäytetyön teon hetkellä vielä projektiin käytettäväksi. Vähäinen määrä yksikäsitteisiä loogisia kokonaisuuksia ja paljon ajuritason integrointia merkitsee, että testauksen kohteita on vaikea löytää. Mikroprosessorin simulaatio tullee olemaan testivetoisen kehityksen tulevaisuus. Seuraava looginen askel on kuitenkin testivetoisen kehityksen prosessin tuonti projektiin, kun laiteintegrointi alkaa vähenemään ja koodin loogisen osuuden määrä kasvamaan.

Sulautettujen järjestelmien kehityksessä käytettävien laitteiden osalta hankalaa oli niiden saatavuus. Käytettäviä laitteita oli rajallisesti ja usein niissä olevat lisenssit rajoittivat käyttöä entisestään. Vaihtoehtona olisi tietenkin ollut avoimeen lähdekoodiin perustuvien debuggereiden valmistaminen esimerkiksi Raspberry Pi -laitteista. Näiden säätäminen toimivaksi kokonaisuudeksi vie kuitenkin aikaa joka on pois hyödyllisestä työstä.

Tuloksena syntyi ohjeet, joiden avulla voi jokainen UnSeenin työntekijä säästää aikaa ottaessaan sulautetun järjestelmän työkalut käyttöön Razr-alustaan liittyvissä projekteissa. Sulautetut järjestelmät tuovat monia poikkeuksia sääntöihin, ja näiden listaaminen helpottanee tulevaa kehitystä.

Vianetsinnässä käytettävien laitteiden ja menetelmien tutkiminen on iteratiivinen prosessi. Kehitys ei pysähdy kirjoitushetkeen, ja uusia vaihtoehtoja tulee

julkisuuteen jatkuvasti. Vaikka tällä hetkellä sulautettujen laitteiden kehitys on vielä aika pitkälle isojen kaupallisten yrityksien käsissä, on avoimeen lähdekoodiin perustuvien ratkaisujen suosio ollut tasaisessa kasvussa. Tätä on edesauttanut muun muassa halpojen mikrokontrollerien tarjonnan kasvu, ja Raspberry Pi -tietokoneiden suosion nousu.

LÄHTEET

Butcher, P. 2009. Debug It! Pragmatic Bookshelf.

Diez, M. 2016. Debugging nRF52 series devices using Trace. Julkaistu 20.3.2016. Luettu 28.6.2019. https://devzone.nordicsemi.com/nordic/nordic-blog/b/blog/posts/debugging-nrf52-series-devices-using-trace

Grenning, J. 2011. Test Driven Development for Embedded C. Pragmatic Bookshelf.

Ingianni, L. 2018. Trying out Jumper Virtual Lab, Part I: getting to blinky. Julkaistu 7.2.2018. Luettu 30.8.2019. https://ingianni.eu/blog/trying_out_jumper_pt1/

nRF52840 Product Specification. 2018. Datalehti. Nordic Semiconductor.

Oshana, R. 2013. Software Engineering for Embedded Systems. 1. Painos. Newnes.

Richter, D. 2014. Flash Memories. Springer Netherlands.

Spinellis, D. 2016. Effective Debugging: 66 Specific Ways to Debug Software and Systems. Addison-Wesley Professional.

Veilleux, D. 2017. Monitor Mode Debugging with J-Link and GDB/Eclipse. Julkaistu 29.4.2017. Luettu 2.7.2019. https://devzone.nordicsemi.com/nordic/nordic-blog/b/blog/posts/monitor-mode-debugging-with-j-link-and-gdbeclipse