Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Viktor Elizarov

Tiedonkeräimen suunnittelu ja toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Sähkö- ja automaatiotekniikka

Insinöörityö

22.10.2019

Tekijä Otsikko Sivumäärä Aika	Elizarov Viktor Tiedonkeräimen suunnittelu ja toteutus 53 sivua + 3 liitettä 22.10.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	sähkö- ja automaatiotekniikka
Ammatillinen pääaine	automaatiotekniikka
Ohjaajat	lehtori Timo Tuominen

Tässä insinöörityössä suunniteltiin ja toteutettiin laite sekä laitteen ohjelmisto, jonka avulla voidaan lukea Wi-Fi-verkon yli laitteeseen liitettyjen sensoreiden ja tulojännitesignaalien mittaustuloksia.

Laite perustuu ARM Cortex-M4F -pohjaiseen mikrokontrolleriin, jonka rinnalla on Wi-Fi- ja Bluetooth-moduulit. Lisäksi laitteessa on microSD-liitäntä sekä USB-protokollan toteuttava sarjaporttimuunnin, jonka avulla on toteutettu laitteen komentorivikonsoli. Laitteeseen voidaan liittää enintään kymmenen I^2C-väyläpohjaista sensoria ja enintään kuusi 24 V -tulojännitesignaalia.

Insinöörityön teoriaosuudessa perehdytään tarkemmin Cortex-M4-prosessoriin ja reaaliaikakäyttöjärjestelmien ominaisuuksiin. Toteutusosuudessa käydään läpi pintapuolisesti laitteen piirilevyn suunnittelu- ja toteutusvaiheet, sekä laitteen ohjelmiston rakenne.

Insinöörityön päätteeksi saatiin toimiva laite ja ohjelmisto, joka toteutti suurimman osan toiminnallisista vaatimuksista.

Avainsanat	STM32, ARM Cortex-M4F, RTOS, sulautetut järjestelmät

Author Title Number of Pages Date	Viktor Elizarov Implementing and designing a Data Logger 53 pages + 3 appendices 22 October 2019
Degree	Bachelor of Engineering
Degree Programme	Electrical and automation engineering
Professional Major	Automation Technology
Instructors	Timo Tuominen, Senior Lecturer

This thesis is about designing and implementing an electronic device, as well as device software. The purpose of the device is to record data over time via external sensors and 24 V digital inputs and enable reading of the recorded data over a Wi-Fi network.

The device is based on ARM Cortex-M4F -series microcontroller, in which it has alongside Wi-Fi and Bluetooth modules, a microSD slot for storing the recorded data. In addition, the device was implemented with a command-line console-based interface that can be used over USB-connection.

The theoretical part of this thesis provides in-depth look at the Cortex-M4 processor and some of the features of real-time operating systems. The implementation section of this thesis concentrates on the designing and manufacturing stages of the device and describes the structure of the program that was developed for this device.

At the end of this project, a functioning device and software was implemented and most of the functional requirements were fulfilled.

Keywords	STM32, ARM Cortex-M4F, RTOS, embedded system

Sisällys

Lyhenteet

1 Johdanto 1

2 ARM Cortex-M4 2

2.1 ARM-prosessorit yleisesti 2
2.2 ARM-prosessorien ja -arkkitehtuuriversioiden nimeämiskäytäntö 2
2.3 Rakenne 3
2.4 Thumb-käskykanta 5
2.5 Rekisterit 5
2.6 ARM EABI -binäärirajapinta 7
2.7 Poikkeuskäsittely 9
2.8 Vektoritaulukko 10
2.9 Poikkeuskäsittelijään siirtyminen 11
2.10 Käyttöjärjestelmätuki 12

3 FreeRTOS-reaaliaikakäyttöjärjestelmä 13

3.1 Yleistä reaaliaikakäyttöjärjestelmistä 13
3.2 Reaaliaikaisuus 13
3.3 Säikeet 14
3.4 Prosessit 16
3.5 Moniajo 16
3.6 Kontekstinvaihto FreeRTOS-käyttöjärjestelmässä 18
3.7 Semaforit 20
3.8 Poissulku 22
3.9 Muistinhallinta FreeRTOS-käyttöjärjestelmässä 24

4 Tiedonkeräimen suunnittelu ja valmistus 27

4.1 Piirilevyn komponentit 27
4.2 Piirikaavion suunnittelu 27
4.3 Piirilevyn layoutin ja komponenttien reititys 32
4.4 Piirilevyn Gerber-tiedostojen luonti ja valmistus 34

5 Ohjelmiston toteutus 36
5.1 Ohjelmointiympäristö 36
5.2 Rakenne 37
5.3 Käynnistys 37
5.4 Säikeet 45
5.5 Konsolikäyttöliittymä 46
6 Yhteenveto 50
Lähteet 51
Liitteet
Liite 1. Komponenttiluettelo
Liite 2. Piirikaavio
Liite 3. Layout

Lyhenteet

ABI *Application Binary Interface.* Ohjelmistojen binäärirajapinta.

API *Application Programming Interface*. Ohjelmointirajapinta.

ARM *Advanced RISC Machines.* Prosessoriarkkitehtuuri.

CTC *Clear to Send.* Sarjaporttiliikenteessä käytettävä vuonohjaussignaali, joka ilmaisee laitteen olevan valmis datan vastaanottoa varten.

DMA *Direct Memory Access.* Tietotekniikassa käytetty väyläarkkitehtuurinen ominaisuus, jonka avulla dataa voidaan siirtää oheislaitteiden mustien välillä ilman, että data käsitellään prosessorin toimesta.

DSP *Digital Signal Processing*. Digitaalinen signaalinkäsittely.

EABI *Embedded Application Binary Interface.* Sulautettujen järjestelmien ohjelmistojen binäärirajapinta.

FPU *Floating Point Unit.* Liukulukuyksikkö (vapaa suomennos).

GCC *GNU Compiler Collection.* GNU-käännöstyökalut.

GDB *GNU Debugger.* GNU-virheenjäljitysohjelma.

GPIO *General-purpose input/output.* Yleiskäyttöisen sisääntulo- tai ulostulolinjan nimitys.

HAL *Hardware Abstraction Layer.* Rajapinta, joka erottaa laitteiston ja ohjelmiston toisistaan.

I^2C *Inter-Integrated Circuit.* Kaksisuuntainen tiedonsiirtoväylä.

LDO	*Low Dropout*. Jänniteregulaattorien yhteydessä käytettävä ilmaisu, kun jänniteregulaattori kykenee toimimaan hyvin pienellä sisään- ja ulostulojännitteen erotuksella.
MPU	*Memory Protection Unit.* Muistinsuojausyksikkö.
MSP	*Main Stack Pointer.* ARM-prosessorin pääpinomuistiosoitin.
NVIC	*Nested Vector Interrupt Controller.* ARM-prosessoreiden järjestelmäkomponentti, joka vastaa poikkeusten käsittelystä.
PendSV	*Pendable Service Call*. Viivästetty ohjelmakeskeytyskutsu.
PSP	*Process Stack Pointer.* ARM-prosessorin prosessipinomuistiosoitin.
RISC	*Reduced Instruction Set Computer.* Suoritinarkkitehtuurin suunnittelufilosofia.
RTS	*Ready to Send.* Sarjaporttiliikenteessä käytettävä vuonohjaussignaali, joka ilmaisee laitteen olevan valmis datan siirtoa varten.
SoC	*System on Chip.* Järjestelmäpiiri.
SVC	*Supervisor Call.* Ohjelmallisen keskeytyspyynnön tuottava konekäsky.
SWD	*Serial Wire Debug.* Prosessorin sarjamuotoinen virheenjäljitysliitäntä.
SWO	*Serial Wire Output.* Prosessorin virheenjäljitysviestien ulostulolinjan nimitys.
UART	*Universal Asynchronous Receiver/Transmitter.* Asynkroninen sarjaliikenteen lähetys-ja vastaanottopiiri.
USB	*Universal Serial Bus.* Yleinen sarjaväyläliitäntä.
Wi-Fi	*Wireless Fidelity.* Langattomien lähiverkkoyhteystuotteiden tavaramerkki.

WIC	*Wakeup Interrupt Controller*. Herätyskeskeytysohjain (vapaa suomennos).

1 Johdanto

Tämän insinöörityön tarkoituksena oli suunnitella ja toteuttaa vähävirtainen tiedonkeruulaite, jonka avulla voidaan tarkkailla ympäristön olosuhteiden muutoksia laitteeseen kytkettävien ulkoisten sensoreiden avulla. Laitteen toiminnallisiin vaatimuksiin kuului, että laitteeseen voidaan liittää enintään kymmenen I^2C-väyläpohjaista sensoria sekä kahdeksan 24 V -tulojännitesignaalia. Lisäksi laitteen tuli toteuttaa mittaustulosten tallennus muistikortille ja mahdollistaa mittaustulosten etälukeminen joko langattoman verkon yli tai USB-väylän kautta ohjelmistoon toteutetun konsolikäyttöliittymän avulla.

Insinöörityön teoriaosuus on jaettu lukuihin 2 ja 3. Raportin toisessa luvussa perehdytään Cortex-M4-prosessorin rakenteeseen ja ominaisuuksiin, sekä kolmannessa luvussa tutustutaan FreeRTOS-reaaliaikakäyttöjärjestelmän ohjelmointirajapintoihin ja toimintamekanismiin. Raportin neljännessä luvussa käsitellään pintapuolisesti laitteiston piirilevyn suunnittelu- ja toteutusosuutta, jonka jälkeen luvuissa 5 käsitellään ohjelmointiosuutta, sekä laitteiston ohjelmiston rakennetta. Lopuksi luvussa 6 tehdään yhteenveto projektin tuloksista.

2 ARM Cortex-M4

2.1 ARM-prosessorit yleisesti

ARM-prosessorit ja -arkkitehtuuri ovat ARM Holdings -nimisen suunnitteluyhtiön tuotteita, joiden kehitys sai alkunsa vuonna 1983 Acorn Computers Limited -nimisen yhtiön toimesta [10, s.43]. Yhtiön ensimmäinen tuote oli vuonna 1981 julkaistu BBC Micro -sarjan tietokoneen rinnakkaissuoritin, jota hyödynnettiin 8-bittisen MOS Technology 6502 -prosessorin rinnalla [9; 10, s. 43].

BBC Micro -sarjan jatkokehityksen yhteydessä etsittiin korvaajaa 6502-prosessorille, jonka seurauksena 32-bittinen ARM-arkkitehtuuri sai alkunsa. ARM-arkkitehtuuria kehitettiin Advanced Research and Development -nimisen osaston toimesta vuodesta 1983 alkaen, jonka kehitystyön tulos oli vuonna 1985 julkaistu ARM1-prosessori. 1990-luvun alussa osasto eriytettiin erilliseksi yhtiöksi ja syntyi Advanced RISC Machines Ltd -niminen yhtiö. [13, s. 36.]

Yhtiön nimi muuttui myöhemmässä vaiheessa ARM Ltd:ksi, kun yhtiön emoyhtiö ARM Holdings listautui pörssiin vuonna 1998.

2.2 ARM-prosessorien ja -arkkitehtuuriversioiden nimeämiskäytäntö

ARM Holdings on vuosien saatossa julkaissut useista eri prosessorimalleja ja ARM-arkkitehtuurin versioita. Samalla myös prosessoreiden nimeämiskäytäntö on muuttunut prosessorimallien kehityksen aikana.

ARM-prosessorimallien kohdalla on tärkeintä ymmärtää prosessorimallin toteuttama ARM-prosessoriarkkitehtuuriversio. ARM-prosessoriarkkitehtuurin versio ilmaistaan ARMv-etuliitteen avulla, jota seuraa arkkitehtuurin pääversion ilmaiseva numero, esimerkiksi ARM arkkitehtuurin 7. versio ilmaistaan nimellä ARMv7. ARMv7-arkkitehtuuriversiota ei tulisi kuitenkaan sekoittaa ARM7-prosessorimallisarjaan, kuten ARM7TDMI-prosessoriin, joka toteuttaa ARMv4T-prosessoriarkkitehtuurin.

ARM-prosessoriarkkitehtuurin pääversionumeroa voi seurata vielä useampi lisäkirjain, jonka avulla ilmaistaan prosessoriarkkitehtuurin laajennukset:

- T-kirjain ilmaisee Thumb-käskykantatuen. Esimerkiksi ARMv4**T**.
- E-kirjain ilmaisee käskykannan laajennuksen digitaalista signaalikäsittelyä varten. Esimerkiksi ARMv5T**E**.
- A-, R- ja M-kirjaimet ilmaisevat prosessoriarkkitehtuurin profiilin. Esimerkiksi ARMv7-**M**. [22.]

Vuodesta 2005 lähtien ARM Holdings on käyttänyt prosessorimallien nimeämisessä Cortex-tuotenimeä, jota seuraa prosessoriarkkitehtuuriprofiilin ilmaiseva lisäkirjain. Esimerkiksi Cortex-M4-prosessorin tapauksessa M-kirjaimella ilmaistaan prosessorin toteuttavan ARMv7-prosessoriarkkitehtuurin M-profiilin. Profiilia ilmaisevaa kirjainta voi myös seurata F-kirjain, joka ilmaisee prosessorin sisältävän liukulukuaritmetiikan toteuttavan apuprosessorin (FPU, engl. Floating Point Unit), esimerkiksi Cortex-M4**F**. [22.]

ARM-prosessoriarkkitehtuurin profiileita hyödynnetään eri käyttötarkoituksissa:

- A-profiilin prosessorit ovat tyypillisesti osana järjestelmäpiiriä (SoC, engl. System on Chip), jotka ovat suunnattu korkeaa suorituskykyä vaativiin laitteisiin [1, s. A1-20].

- R-profiilin prosessoreita hyödynnetään korkeaa vikasietoisuutta vaativissa laitteissa, esimerkiksi teollisuuden laitteissa [1, s. A1-20].

- M-profiilin prosessorit ovat suunnattu mikrokontrollerikäyttöön, joissa deterministisyys ja vähäinen virrankulut ovat suorituskykyä tärkeämpi ominaisuus [1, s. A1-21].

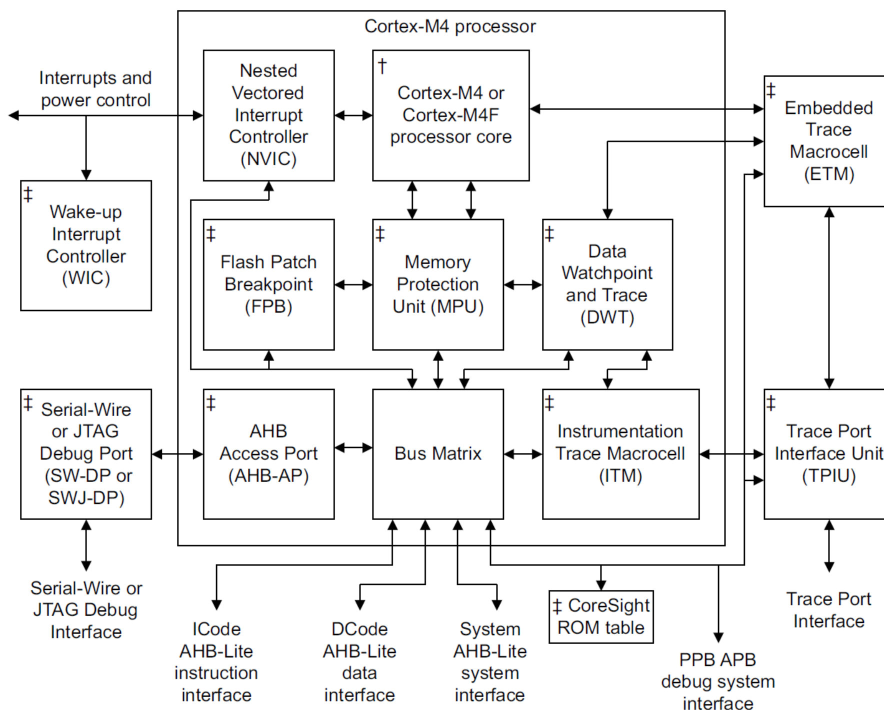
2.3 Rakenne

Cortex-M4 on vuonna 2010 julkaistu 32-bittinen prosessori, joka on suunnattu mikrokontrollerikäyttöön. Cortex-M4 toteuttaa ARMv7E-M-prosessoriarkkitehtuurin ja tukee 16-bittistä Thumb-käskykantaa sekä Thumb-2-teknologian tuomia 32-bittisiä käskyjä. Lisäksi Cortex-M4-prosessori toteuttaa Harvard-arkkitehtuurin mukaisen erillisen käsky- ja

datamuistiväylän, joka mahdollistaa rinnakkaisaikaisen konekäskyn ja datan siirron, mikä tuottaa korkeamman suorituskyvyn.

Cortex-M4-prosessorissa on digitaaliseen signaalinkäsittelyyn erikoistunut apuprosessori (DSP, engl. Digital Signal Prosessing) sekä vaihtoehtoisesti liukulukuaritmetiikan toteuttava apuprosessori, josta Cortex-M4F-mallin F-kirjain juontaa juurensa. Lisäksi Cortex-M4-prosessoriin on saatavilla vaihtoehtoinen muistinsuojausyksikkö (MPU, engl. Memory Protection Unit), jonka avulla voidaan rajata suoritettavan ohjelmakoodin pääsyä tiettyihin muistialueisiin.

Kuva 1 havainnollistaa Cortex-M4-prosessorin lohkokaavioesityksen, josta käy ilmi prosessorin järjestelmäkomponentit. Prosessorin rakenteen sisäisistä komponenteista olennaisin on keskeytysohjain (NVIC, engl. Nested Vector Interrupt Controller) sekä vaihtoehtoinen herätyskeskeytysohjain (WIC, Wakeup Interrupt Controller) jonka avulla voidaan toteuttaa keskeytysohjattuja virransäästöominaisuuksia.



Kuva 1. Cortex-M4 prosessorin lohkokaavioesitys [2, s. 2-21].

2.4 Thumb-käskykanta

Thumb-käskykanta on vuonna 1995 ARM7TDMI(R)-prosessorin yhteydessä esitelty 16-bittinen käskykanta, joka toteuttaa osajoukon ARM-käskykannan toiminnallisuudesta [14, s. 87]. ARMv4T-prosessoriarkkitehtuurista lähtien ARM-käskykantaan pohjautuvat prosessorit kykenevät suorittamaan sekä Thumb- että ARM-käskykannasta tuotettuja ohjelmia sillä poikkeuksella, että prosessorin suoritustila on vaihdettava ohjelmansuorituksen aikana Thumb-tilaan, koska ARM- ja Thumb-käskyt eivät ole keskenään yhteensopivia [13, s. 189].

Thumb-käskykannasta tuotettu ohjelma on keskimäärin 30 % pienempi kuin vastaavanlainen ARM-käskykannasta tuotettu ohjelma [14, s. 87]. Etuna on, että korkeaa suorituskykyä vaativat ohjelmatoiminnallisuudet voidaan toteuttaa ARM-käskykantaa käyttäen ja ei-suorituskykykriittiset toiminnallisuudet Thumb-käskykannalla, joka parhaimmillaan johtaa pienempään ohjelmakokoon suorituskyvystä tinkimättä.

Vuonna 2003 ARM Holdings julkisti ARMv6T2-prosessoriarkkitehtuurin yhteydessä Thumb-2-teknologian, joka laajensi Thumb-käskykannan toiminnallisuusuuden ARM-käskykannan tasolle uusien 32-bittisten Thumb-käskyjen avulla [14, s. 87]. Thumb-2-teknologian avulla prosessori kykenee suorittamaan Thumb- sekä ARM-konekäskyjä saman suoritustilan aikana [19, s.156].

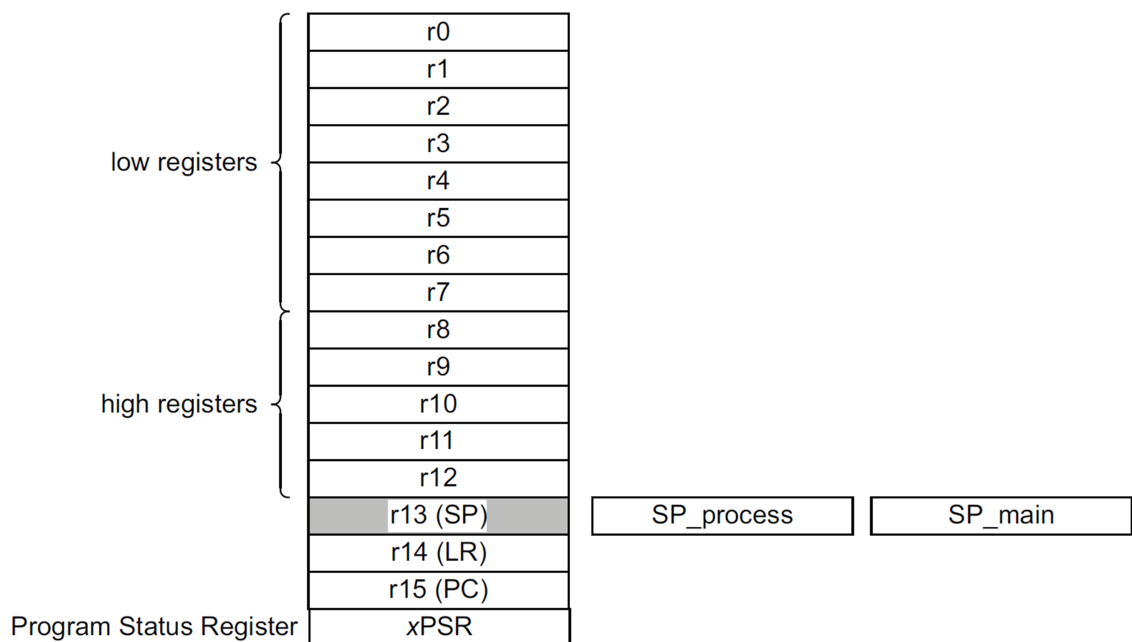
2.5 Rekisterit

ARM-prosessoriarkkitehtuuri noudattaa RISC-tyyppisille arkkitehtuureille ominaista load-store-suoritinarkkitehtuurimallia, jossa konekäskyjä käsitellään vain prosessorin sisäisten rekistereiden avulla. Käyttömuistiin viitataan vain load- ja store-käskyillä, joiden avulla kopioidaan dataa prosessorin rekistereiden ja järjestelmämuistin välillä [13, s. 41].

Cortex-M4-prosessorissa on yhteensä 21 kappaletta 32-bittistä rekisteriä, joista ensimmäiset 16 rekisteriä ovat niin kutsuttuja ydinrekistereitä, joihin viitataan myös englanninkielisellä nimityksellä "ARM Core Registers". Loput viisi rekisteriä ovat erikoisrekistereitä, jotka liittyvät prosessorin poikkeuskäsittelyihin, virheidenilmaisuun sekä prosessorin käyttötilojen hallintaan. [1, s. B1-572.]

Kuva 2 havainnollistaa Cortex-M4-prosessorin rekistereiden järjestyksen.

- Rekisterit r0–r12 ovat yleiskäyttöisiä rekistereitä, jotka ovat varattu ohjelman suoritukseen liittyvän datan, käskyjen ja osoitteiden käsittelyä varten. Huomioitavaa on, että 16-bittisiä Thumb-konekäskyjä voidaan käsitellä vain rekistereissä r0–r7. Poikkeuksena ovat käskyt MOV, CMP ja ADD, joita voidaan operoida rekistereissä r8–r12.
- Rekisteri r13 viittaa pinomuistiin, mutta muista rekistereistä poiketen kyseinen rekisteri on kahdennettu (engl. banked). Riippuen suorittimen käyttötilasta, rekisteri viittaa joko pääpinomuistiosoittimeen (MSP, engl. Main Stack Pointer) tai prosessipinomuistiosoittimeen (PSP, engl. Process Stack Pointer). Rekisterin kahdennuksen taustalla on mahdollisen käyttöjärjestelmän pinomuistin eristäminen sovellusohjelman pinomuistista suorittimen käyttötilojen avulla.
- Rekisteri r14 (LR, engl. Link Register) pitää sisällään osoitteen, johon palataan aliohjelman suorituksen jälkeen. Tämä ei koske poikkeuskäsittelyitä, jolloin kyseistä rekisteriä käytetään toimintatilan tallentamiseen, johon prosessorin on palattava poikkeuskäsittelyn jälkeen [8, s. 2-28].
- Rekisteri r15 (PC, engl. Program Counter) pitää sisällään seuraavaksi suoritettavan konekäskyn osoitteen.
- Loput rekisterit ovat erikoisrekistereitä. Ydinrekistereistä poiketen, erikoisrekistereitä voidaan käsitellä vain MSR- ja MRS-konekäskyjen avulla.



Kuva 2. Cortex-M4-prosessorin rekisterit [2, s. 3-47].

2.6 ARM EABI -binäärirajapinta

Binäärirajapinta (ABI, engl. Application Binary Interface) on standardi, joka määrittelee prosessoriarkkitehtuurin kutsumallin sekä toimintamallin kahden ohjelman tai ohjelman ja ohjelmakirjaston välillä [20, s. 81]. Sulautettujen järjestelmien ohjelmistot ovat yleensä kuitenkin täysin itsenäisiä kokonaisuuksia, eivätkä hyödynnä esimerkiksi dynaamisesti linkitettyjä ohjelmakirjastoja. Tällöin sovelletaan sulautetun järjestelmän binäärirajapintaa (EABI, engl. Embedded Application Binary Interface).

Kutsumalli (engl. calling convention) on osa binäärirajapintastandardia ja sen avulla määritellään, miten käännetyn ohjelmakoodin tulee toteuttaa aliohjelmien kutsu, sekä parametrien ja paluuarvon välitysmekanismi [20, s. 81]. Toisin sanoen kutsumalli määrittelee, miten ohjelmointikielikääntäjän tulisi tuottaa ohjelmakoodista prosessoriarkkitehtuurin mukaisia konekäskyjä. ARM-arkkitehtuurin kutsumalli on määritelty dokumentaatiossa nimeltä “Procedure Call Standard for the ARM Architecture (AAPCS)”. Kutsumallin lisäksi dokumentaatio määrittelee pinokehykseen liittyviä ominaisuuksia, yleiskäyttöisten rekistereiden toiminnallisuuden sekä C-ohjelmointikielessä esiintyvien tietotyyppien koon.

Vaikka sulautettujen järjestelmien ohjelmistot ovat yleensä täysin itsenäisiä kokonaisuuksia, on binäärirajapinnan noudattaminen suositeltavaa hyödynnettäessä matalan tason ohjelmointikieliä, kuten ARM Assembly -kieltä. Esimerkiksi Cortex-M4-prosessorin kontekstinvaihdon yhteydessä tapahtuva pinokehyksen tallentaminen suoritetaan NVIC:in toimesta. NVIC on suunniteltu sillä oletuksella, että mikrokontrollerissa suoritettava ohjelmisto noudattaa ARM EABI -kutsumallia, jolloin NVIC tallentaa pinokehykseen vain kutsumallissa määriteltyjen rekistereiden arvot.

Onneksi nykyaikaiset ohjelmointikielikäännöstyökalut ovat sen verran hyviä, että korkean tason ohjelmointikieliä käytettäessä ei tarvitse tuntea edes tuntea binäärirajapinta-termiä, kunhan vain kykenee valitsemaan oikeat käännöstyökalut. Ohjelmoijan olisi kuitenkin hyödyllistä sisäistää kutsumallin mukaisen rekistereiden käytön aliohjelman kutsun yhteydessä, mikäli tavoitteena on RISC-tyyppiselle prosessorille tuotetun ohjelmakoodin optimointi suoritusnopeuden tai ohjelman koon osalta.

RISC-tyyppisissä prosessoriarkkitehtuureissa nopeuden optimoinnin kannalta on tärkeää, että prosessorin rekistereitä hyödynnetään mahdollisimman paljon. ARM-kutsumallin tapauksessa aliohjelman ensimmäiset neljä parametria välitetään rekistereiden r0–r3 kautta [5, s. 15], ylimenevät parametrit välitetään pinon kautta. Näin ollen optimaalisinta olisi, että aliohjelman ensimmäiset parametrit ovat myös aliohjelman suorituksen kannalta ensimmäiset neljä tarvittavaa parametria, muutoin ne kopioidaan pinomuistiin aliohjelman alussa, mikäli kyseisiä rekistereitä hyödynnetään muuhun tarkoitukseen.

ARM-kutsumallista ilmenee myös, että parametrit välitetään aina 4 tavun kohdistuksella. Seurauksena on, että rekisterin kautta välitettävät parametrit, joiden tietotyyppi on lyhyempi kuin prosessorin sanan pituus, laajennetaan aina prosessorin sanan pituiseksi. Lisäksi tuplasanan (engl. double word) pituiset parametrit välitetään kahden rekisterin avulla, esimerkiksi rekisterin r0 ja r1 tai r2 ja r3 [5, s. 15.] Tämä tarkoittaa sitä, että aliohjelman parametrit tulisi järjestellä siten, että tuplasanan pituinen parametri on ensimmäisenä tai kolmantena parametrina, muulloin parametrit välitetään pinomuistin kautta.

Havainnollistetaan edellä mainitut ARM-kutsumallin säännöt todeksi kääntämällä esimerkkikoodin 1 C-ohjelma Thumb-käskyiksi ilman kääntäjän suorittamaa optimointia. Käytetään ARM EABI -binäärirajapintaa varten tarkoitettua GNU GCC 8.3.1 -ohjelmointikielikääntäjää kääntäjäparametrien -mthumb, -mcpu=cortex-m4 ja -O0 avulla.

```
#include <stdint.h>

static void foo(int32_t   a,
                long long b,
                int32_t   c) { /* ... */ }

static void bar(long long b,
                int32_t   a,
                int32_t   c) { /* ... */ }


int main(void)
{
    foo(20, 40, 60);
    bar(40, 20, 60);

    return(0);
}
```

Esimerkkikoodi 1. Aliohjelman parametrien välittämistä havainnollistava C-ohjelmakoodi.

Tutkimalla esimerkkikoodin 1 foo- ja bar-aliohjelmien kutsusta tuotettuja Thumb-konekäskyjä voidaan havaita, että foo-aliohjelmakutsun c-argumentti välitetään pinon kautta: Aliohjelmalle välitettävä arvo 60 siirretään ensiksi rekisteriin r3, jonka jälkeen rekisterin r3 arvo kopioidaan pinomuistiin. Sen sijaan bar-aliohjelmakutsun kaikki kolme argumenttia välitetään rekistereiden r0–r3 avulla.

```
main:
    push  {r7, lr}
    sub   sp, sp, #8
    add   r7, sp, #8
    movs  r3, #60
    str   r3, [sp]
    mov   r2, #40
    mov   r3, #0
    movs  r0, #20
    bl    foo
    movs  r3, #60
    movs  r2, #20
    mov   r0, #40
    mov   r1, #0
    bl    bar
    movs  r3, #0
    mov   r0, r3
    mov   sp, r7
    pop   {r7, pc}
```

Esimerkkikoodi 2. Esimerkkikoodi 1:stä tuotetut Thumb-konekäskyt.

2.7 Poikkeuskäsittely

ARM-prosessoriarkkitehtuurissa poikkeus on määritelty tapahtumana, joka muuttaa ohjelman normaalia kulkuvirtaa [1, s. B1-569], eli toisin sanoen tapahtuma, jonka takia ohjelmansuoritus joudutaan keskeyttämään. ARMv7E-M-arkkitehtuurissa poikkeuksiin luokitellaan järjestelmälaitteiden keskeytykset, virheiden käsittely sekä ohjelmalliset keskeytyspyynnöt [1, s. B1-569]. Termin nimestä huolimatta poikkeukset ovat oleellinen osa prosessorin toiminnallisuutta ja ovat täysin odotettuja järjestelmän ohjelmansuorituksen kannalta, virhetilanteita lukuun ottamatta.

Poikkeukset ovat erityisen tärkeässä roolissa hyödynnettäessä asynkronisia ohjelmointimenetelmiä. Tämmöisiä tilanteita voivat olla esimerkiksi prosessorin ja DMA-ohjaimen välinen kommunikointi, jossa prosessori odottaa asynkronisesti DMA-ohjaimelta tiedonsiirron päätymistä ilmaisevaa keskeytystä. Tällöin perinteisen odotusmenetelmän sijaan prosessorin suoritinaikaa voidaan hyödyntää johonkin muuhun operaatioon.

Taulukko 1 havainnollistaa ARMv7E-M-prosessoriarkkitehtuurin poikkeustyypit ja niiden aktivointimenetelmät, jotka ovat joko synkronisia tai asynkronisia. Synkronisiin poikkeuksiin luokitellaan tietyt järjestelmävirheet sekä ohjelmallinen keskeytyspyyntö, jonka avulla siirrytään etuoikeutettuun suoritustilaan. Muissa tapauksissa poikkeus luokitellaan asynkroniseksi ja käsitellään jonotusperiaatteella poikkeuksen prioriteetin mukaan.

Taulukko 1. ARMv7E-M-prosessoriarkkitehtuurin poikkeukset [8, s. 2-22].

Poikkeustyyppi	**Prioriteetti**	**Poikkeusnumero**	**Aktivointi**
Reset	-3 (Korkein)	1	Asynkroninen
NMI	-2	2	Asynkroninen
HardFault	-1	3	-
MemManage	Konfiguroitavissa	4	Synkroninen
BusFault	Konfiguroitavissa	5	Synkroninen
UsageFault	Konfiguroitavissa	6	Synkroninen
SVCall	Konfiguroitavissa	11	Synkroninen
PendSV	Konfiguroitavissa	14	Asynkroninen
SysTick	Konfiguroitavissa	15	Asynkroninen
Interrupt (IRQ)	Konfiguroitavissa	16 + n	Asynkroninen

2.8 Vektoritaulukko

Vektoritaulukko on nimensä mukaisesti taulukko, joka koostuu poikkeusvektoreista (engl. exception vector). Poikkeusvektorilla tarkoitetaan taulukon alkiota, joka pitää sisällään poikkeuksen käsittelyyn vaadittavan ohjelmakoodin osoitteen, eli poikkeuskäsittelijän (engl. exception handler). Vektoritaulukon alkiot ovat kooltaan prosessorin sanan pituisia, eli Cortex-M4-prosessorin tapauksessa alkiot ovat 32 bitin pituisia. [19, s. 270.]

Vektoritaulukko sijaitsee järjestelmän ohjauslohkon (SCB, engl. System Control Block) VTO-rekisterin osoittamassa osoitteessa, jonka arvo on prosessorin nollauksen jälkeen nolla [1, s. B3-657]. Tämä tarkoittaa sitä, että vektoritaulukko on sijoitettava ohjelmamuistin alkupäähän, mutta mikäli on tarpeen, VTO-rekisterin arvoa voidaan muuttaa ohjelmansuorituksen aikana, esimerkiksi jos vektoritaulukko halutaan siirtää nopeammalle muistille poikkeuskäsittelyn aiheuttaman viiveen pienentämiseksi.

Jokaiselle poikkeustyypille on määritelty oma poikkeusnumero (engl. exception number), joka ilmaisee poikkeusvektorin indeksin vektoritaulukossa. ARMv7E-M-prosessoriarkkitehtuurissa on käytössä yhteensä 240 poikkeusnumeroa, joista poikkeusnumerot 0–15

on varattu arkkitehtuuriominaisille poikkeusvektoreille ja pinomuistin alkuosoitetta varten. Loput poikkeusnumerot ovat tarkoitettu NVIC:in ulkoisia keskeytyslinjoja varten, joiden toteutus tapahtuu ARM-prosessorin lisensoivan mikropiirivalmistajan toimesta. Tyypillisesti mikropiirivalmistaja integroi ulkoisiin keskeytyslinjoihin oheisjärjestelmäkomponenttien, kuten DMA- ja GPIO-ohjainten keskeytyssignaalit.

2.9 Poikkeuskäsittelijään siirtyminen

Cortex-M4-prosessorin poikkeuskäsittelijään siirtyminen tapahtuu useamman eri vaiheen kautta. Ensimmäinen vaihe on pinokehyksen tallentaminen, jonka aikana pinomuistiin kopioidaan rekistereiden r0–r3, r12, LR, PC ja xPSR sisältö [1, s. B1-587]. Kyseisen operaatio tunnetaan myös termillä kontekstinvaihto (engl. context switch) [20, s. 103].

Mikäli kyseessä on liukulukuaritmetiikan toteuttava Cortex-M4F-prosessori, suoritetaan laajennettu pinokehyksen tallennus. Pinomuistista varataan liukulukuyksikön rekistereitä varten ylimääräiset 128 tavua, mutta rekistereiden sisältöä ei kopioida pinomuistiin, ellei poikkeuskäsittelyn aikana hyödynnetä liukulukulaskentaa. Kyseistä ominaisuutta kutsutaan ARM-arkkitehtuurissa nimellä “Lazy Stacking”, jonka tarkoituksena on pienentää kontekstinvaihdon aiheuttamaa viivettä. [21.]

Sillä välin, kun prosessori suorittaa kontekstinvaihtoa, prosessori noutaa vektoritaulukosta aktiivisen poikkeuksen poikkeusvektorin ja samalla päivittää rekisterin r14 (LR) arvon, johon tallennetaan prosessorin poikkeusta edeltänyt toimintatila [1, s. B1-595]. Normaalin ohjelmansuorituksen aikana kyseinen rekisteri pitää sisällään ohjelmakoodin paluuosoitteen aliohjelmakutsun yhteydessä.

Mikäli ennen poikkeuskäsittelystä pois siirtymistä ilmenee, että jonossa on uusi poikkeus, siirtyy prosessori käsittelemään jonossa olevaa poikkeusta normaaliin ohjelmansuoritukseen siirtymiseen sijasta. Tätä ominaisuutta kutsutaan ARM-prosessoriarkkitehtuurissa nimellä “Tail Chaining”, jonka avulla ketjutetaan peräkkäisiä poikkeuskäsittelyitä. Kyseinen ominaisuus pienentää poikkeuskäsittelyiden viivettä ohittamalla poikkeusten välisen kontekstinvaihdon. [1, s. B1-604.]

2.10 Käyttöjärjestelmätuki

ARMv7E-M-arkkitehtuurissa on useita eri ominaisuuksia, jotka kaikki yhdessä muodostavat tuen reaaliaikakäyttöjärjestelmän toimintamekanismille:

- 24-bittinen järjestelmäajastin ja ajastimen SysTick-järjestelmäpoikkeus, joka luo säännöllisen aikapohjan käyttöjärjestelmän ytimen vuorottajalle [1, s. B3-676]
- synkroninen ohjelmallisen keskeytyspyynnön tuottava konekäsky (SVC) [1, s. A2-33]
- viivästetty ohjelmakeskeytyskutsu (PendSV) [1, s. A2-33]
- kahdennettu pino: prosessipino-osoitin (PSP) ja pääpino-osoitin (MSP) [1, s. B1-572]
- kaksi eri käyttötilaa: säietila (engl. thread mode) sekä käsittelijätila (engl. handler mode) [1, s. B1-568]
- kaksi eri suoritustilaa: etuoikeutettu suoritustila (engl. privileged mode) ja etuoikeudeton (engl. unprivileged mode) [1, s. B1-568].

Lisäksi ARMv7E-M-arkkitehtuuri tarjoaa vaihtoehtoisen muistinsuojausyksikön, jonka avulla voidaan rajoittaa etuoikeudettoman ohjelmakoodin pääsyn rajattuihin muistialueisiin [19, s. 33]. Kuvan 3 taulukko havainnollistaa Cortex-M4-prosessorin kolme eri toimintatilaa, jotka muodostuvat käyttötiloista, suoritusoikeuksista sekä kahdennetusta pinosta.

Table B1-1 Mode, privilege and stack relationship

Mode	Privilege	Stack pointer	Typical usage model
Handler	Privileged	Main	Exception handling.
Thread	Privileged	Main	Execution of a privileged process or thread using a common stack in a system that only supports privileged access.
		Process	Execution of a privileged process or thread using a stack reserved for that process or thread in a system that only supports privileged access, or that supports a mix of privileged and unprivileged threads.
Thread	Unprivileged	Main	Execution of an unprivileged process or thread using a common stack in a system that supports privileged and unprivileged access.
		Process	Execution of an unprivileged process or thread using a stack reserved for that process or thread in a system that supports privileged and unprivileged access.

Kuva 3. ARMv7-M prosessoriarkkitehtuurin toimintatilat [1, s. B1-568].

3 FreeRTOS-reaaliaikakäyttöjärjestelmä

3.1 Yleistä reaaliaikakäyttöjärjestelmistä

Reaaliaikakäyttöjärjestelmä on ohjelma, jonka päätehtävä on ajoittaa ohjelmansuoritusta reaaliaikavaatimusten puitteissa [26, s. 65]. Reaaliaikaisuudella ei viitata käyttöjärjestelmän nopeuteen, vaan että käyttöjärjestelmän vuorottaja (engl. scheduler) noudattaa determinististä suorituskuviota, eli käyttöjärjestelmän toiminta on oltava ennakoitavissa ja vasteajat tunnettuja [27, s. 115]. Tyypillisesti reaaliaikakäyttöjärjestelmän determinististisyys on saavutettu priorisoivan vuorotusalgoritmin ja viiveellisesti rajallisten järjestelmäkutsujen avulla, joiden minimi- ja maksimiarvot ilmoitetaan prosessorin kellopulssijaksojen muodossa.

Nimityksistään huolimatta, todellisuudessa useimmat reaaliaikakäyttöjärjestelmät koostuvat vain ytimestä (engl. kernel). Ohjelmakooltaan reaaliaikakäyttöjärjestelmät ovat yleensä muutamien kilotavujen luokkaa, jonka takia reaaliaikakäyttöjärjestelmiä hyödynnetään laajasti sulautetuissa järjestelmissä – myös järjestelmissä, joissa ei ole reaaliaikavaatimuksia.

Tässä luvussa tutustutaan FreeRTOS-reaaliaikakäyttöjärjestelmään, jota hyödynnetään tämän projektin ohjelmistossa, sekä kyseisen käyttöjärjestelmän 10.0-version rajapintoihin ja ytimen vuorotusmekanismin toimintaan Cortex-M4-prosessorissa.

3.2 Reaaliaikaisuus

Reaaliaikaisuus on avainominaisuus aika- ja turvakriittisissä käyttökohteissa, joissa ohjelmiston tehtäville on asetettu tarkat määräajat. Kyseiset tehtävät saattavat olla esimerkiksi tiettyyn signaaliin tai keskeytykseen reagointi taikka tietyn määräajan saavuttaminen. Riippuen reaaliaikaisuuden tulkintatavasta reaaliaikaisuus jaetaan kahteen tai kolmeen eri kategoriaan:

- Kova reaaliaikaisuus (engl. hard real-time). Tehtävällä on ensimmäinen ja viimeinen ehdoton toiminta-aika, jonka ohitus johtaa järjestelmän pettämiseen [25, s. 28].
- Pehmeä reaaliaikaisuus (engl. soft real-time). Tehtäville on asetettu tarkat toiminta-ajat, joiden ohitukset sallitaan, sillä ehdolla, että tehtävät

suoritetaan ja suoritetun tehtävän lopputuloksella on edelleen arvoa tehtävän suorituksen jälkeen [25, s. 28].

- Kiinteä reaaliaikaisuus (engl. firm real-time). Järjestelmä voi ohittaa tehtävän suorituksen kokonaan, mikäli tehtävään ei voida reagoida määräaikaan mennessä [28, s. 7].

3.3 Säikeet

Säie (engl. thread) on ohjelmarunko, jonka suoritusta hallinnoidaan käyttöjärjestelmän ytimen vuorottajan toimesta. Mikäli ohjelmansuorituksen aikana halutaan hyödyntää useampaa säiettä, eli toisin sanoen suorittaa moniajoa, on säikeen ohjelmakoodin sisällytettävä synkronointipiste, jonka avulla suoritusvuoro voidaan vapauttaa toiselle säikeelle. Synkronointipisteellä viitataan ohjelmakoodin kohtaan, jossa odotetaan tiettyä asynkronista tapahtumaa, esimerkiksi järjestelmäkomponentilta tulevaa signaalia, tietyn resurssin vapautumista tai toiselta säikeeltä tulevaa viestiä. [28, s. 485.]

FreeRTOS-käyttöjärjestelmän säikeiden elinkaari ja toimintamekanismi ei juurikaan eroa muista käyttöjärjestelmistä. Säikeen elinkaaren alkupuolella, jokaista säiettä kohden varataan kontrollilohko, johon sisältyy säikeelle varattu pinomuisti. Kyseistä kontrollilohkoa kutsutaan myös säikeen kontekstiksi ja sitä käytetään ytimen toimesta apuna säikeiden vuorottamisessa. Säikeen pinomuisti on erillään järjestelmän pinomuistista, ja se varataan järjestelmän kekomuistista (engl. heap memory), tosin FreeRTOS-käyttöjärjestelmä tarjoaa säikeen alustukselle vaihtoehtoisen rajapinnan, jonka avulla säikeen pinomuisti voidaan varata jo ohjelman käännösvaiheessa käyttäen staattista muistinvarausta.

FreeRTOS-käyttöjärjestelmässä säikeiden määrä rajoittaa ainoastaan käyttömuistin määrä. Tästä huolimatta säikeitä tulisi käyttää harkiten monista eri syistä, jotka kerrotaan tässä luvussa. Yksi näistä syistä on ohjelmansuorituksen hidastuminen käytettäessä irrottavaa vuorottamisalgoritmia, sillä vuorottaja suorittaa säikeiden suoritusvuoron vaihtumisen aikana kontekstinvaihdon, jossa säikeen tila tallennetaan pinomuistiin ja palautetaan suoritusvuoron saavan säikeen tila pinomuistista [28, s. 405]. Kontekstinvaihto on yleisesti ottaen nopea operaatio, mutta jos säikeiden suoritusvuoroa vaihdetaan tiuhaan tahtiin, syntyy väistämättä viivettä ohjelmansuoritukseen.

Säikeiden luonti FreeRTOS-käyttöjärjestelmässä tapahtuu määrittelemällä säikeelle pinomuistin koko, prioriteetti, säikeelle argumenttina välitettävä osoitin, sekä ohjelmakoodin runko. Esimerkkikoodi 3 havainnollistaa säikeen luontia FreeRTOS-käyttöjärjestelmän ohjelmointirajapintaa käyttäen sekä signalointia keskeytyskäsittelijästä käyttäen keskeytyskäsittelijäturvallista rajapintaa.

```
#include <FreeRTOS.h>
#include <task.h>
#include <stdint.h>

#define INTERRUPT_SIGNAL (1 << 1U)

static TaskHandle_t t_hand;

static void thread_0(void* argument);
{
     BaseType_t rc;
     uint32_t   signal;

     while ( 1 )
     {
          // Wait signal for an indefinite amount of time.

          rc= xTaskNotifyWait(0, INTERRUPT_SIGNAL, &signal, portMAX_DELAY);
          if ( rc == pdTRUE && (signal & INTERRUPT_SIGNAL) )
          {
              handle_signal();
          }
     }
}

int main(void)
{
     BaseType_t rc;

     // Create example thread for interrupt handling.

     rc= xTaskCreate(thread_0, “t_0”, 256, tskIDLE_PRIORITY + 1, & t_hand);
     If ( rc != pdTRUE )
          return(-1);

     // Start scheduler.

     vTaskStartScheduler();

     return(0);
}

void interrupt_1(void)
{
     BaseType_t t_woken = pdFALSE;

     if ( t_hand )
     {
         // Notify example thread 0.

         xTaskNotifyFromISR(t_hand, INTERRUPT_SIGNAL, eSetBits, &t_woken);

         // Request context switch if higher priority task was woken.

         portYIELD_FROM_ISR(t_woken);

     }
}
```

Esimerkkikoodi 3. Säikeen luonti ja signalointi FreeRTOS-rajapintoja käyttäen.

3.4 Prosessit

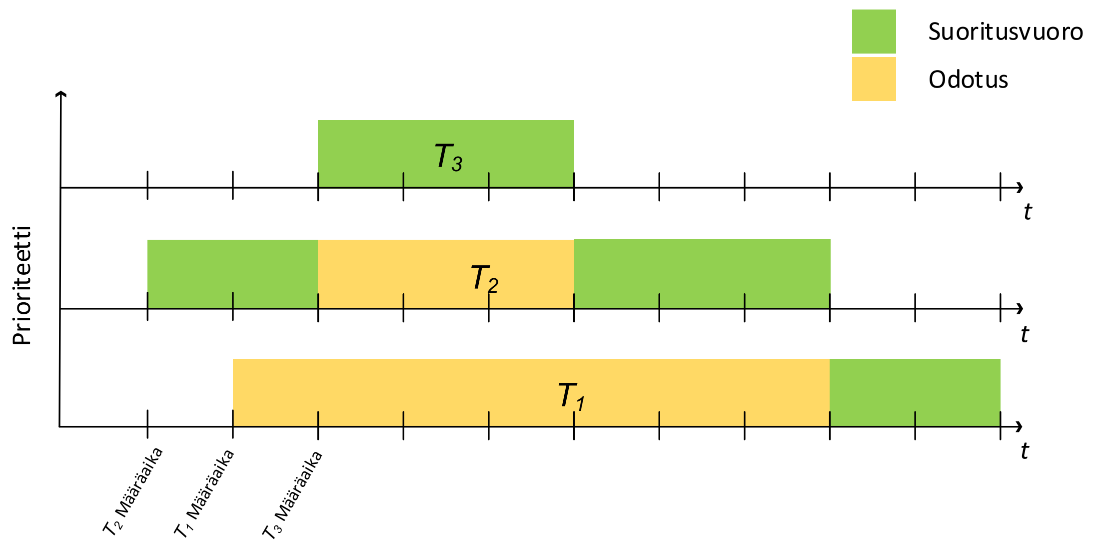
POSIX-standardikokonaisuudessa termillä prosessi tarkoitetaan ohjelmainstanssia, joka koostuu yhdestä tai useammasta säikeestä. Prosesseilla on oma eristetty muistiavaruus ja kommunikointi muiden prosessien kesken tapahtuu vain käyttöjärjestelmän ytimen kautta. Unix-tyyppisissä käyttöjärjestelmissä ohjelman käynnistyksen yhteydessä luodaan prosessi, jolla on alustavasti yksi säie, josta ohjelmansuoritus alkaa. Sulautettujen järjestelmien yhteydessä puhutaan harvemmin prosesseista, sillä eristetty muistiavaruus vaatii virtuaalisen muistinhallinnan, mutta toisaalta sulautetun järjestelmän ohjelmistoa kokonaisuutena voidaan tulkita yhdeksi prosessiksi.

3.5 Moniajo

Moniajolla viitataan käyttöjärjestelmän ominaisuuteen, jonka avulla voidaan suorittaa useampaa säiettä rinnakkaisesti [28, s. 369]. Yhden prosessoriytimen järjestelmissä moniajo toteutetaan suoritinajan viipaloinnilla eri säikeiden kesken, eli toisin sanoen vaihtelemalla säikeiden suoritusvuoroa. Vuorottamisesta vastaa ytimen vuorottaja, jonka toiminta perustuu tiettyyn vuorottamisalgoritmiin. Vuorottamisalgoritmi on tyypillisesti käyttöjärjestelmäkohtainen, ja yleisimmät algoritmityypit ovat

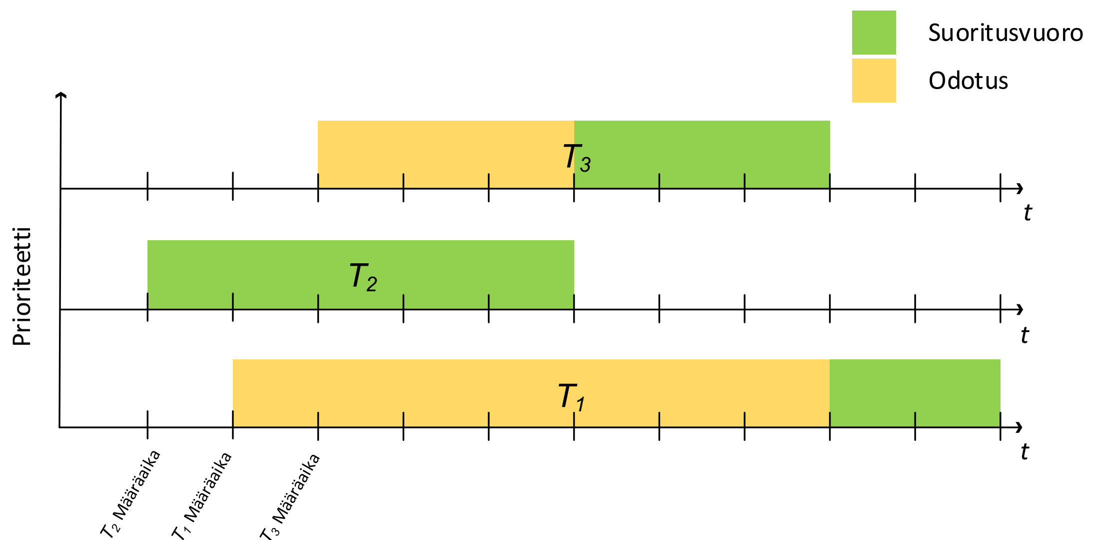
- irrottava (engl. preemptive)
- vuorotteleva (engl. cooperative)
- kiertovuorotteleva (engl. round-robin) [10, s. 23].

Edellä mainituista algoritmeista yleisin käytetty on irrottava, jossa suoritinaikaa jaetaan säikeiden prioriteettien perusteella. Korkeimman prioriteetin omaava säie, joka on valmiina ohjelmansuoritusta varten, saa suoritusvuoron (kuva 4). Säiettä suoritetaan niin kauan, kunnes korkeamman prioriteetin säie on valmis suoritusta varten tai säie saavuttaa synkronointipisteen eli joutuu odottavaan tilaan.



Kuva 4. Irrottava vuorottamisalgoritmi.

Vuorotteleva algoritmi eroaa irrottavasta siltä osin, että suoritusvuoron vapautus on suoritusvuorossa olevan säikeen vastuulla, jolloin ohjelmansuoritus voi jatkua, vaikka korkeamman prioriteetin säie olisi jo saavuttanut määräajan (kuva 5) [10, s. 24].



Kuva 5. Vuorotteleva vuorottamisalgoritmi.

Kiertovuorottelevassa algoritmissa säikeitä suoritetaan vuorotellen tietyn kiinteän aikamäärän verran. Kyseinen algoritmi perustuu siihen, että kaikilla säikeillä on yhtä suuri prioriteetti, eikä näin ollen myöskään täytä reaaliaikavaatimuksia. [10, s. 24.]

3.6 Kontekstinvaihto FreeRTOS-käyttöjärjestelmässä

Termillä konteksti tarkoitetaan käyttöjärjestelmien yhteydessä säikeen ohjelmansuoritukseen liittyvää dataa, joka vaaditaan, mikäli keskeytetyn säikeen ohjelmansuoritus halutaan palauttaa keskeytystä edeltäneeseen tilaan [20, s. 102]. Kontekstinvaihto suoritetaan käyttöjärjestelmän ytimen toimesta säikeen suoritusvuoronvaihdon yhteydessä. Huomioitavaa on kuitenkin se, että termin nimestä huolimatta käyttöjärjestelmän kontekstinvaihto ei liity millään tavalla prosessorin suorittamaan kontekstinvaihtoon, vaikka operaatiossa tallennetaan prosessorin rekistereiden arvot.

Käyttöjärjestelmän suorittama kontekstinvaihto tapahtuu ytimen toimesta vuorottamisen yhteydessä. Käyttöjärjestelmän vuorotusmekanismin tueksi ARMv7E-M–prosessoriarkkitehtuurissa on kolme erilaista järjestelmäpoikkeusta, joiden varaan FreeRTOS-käyttöjärjestelmän vuorotus- ja kontekstinvaihtomekanismi on rakennettu. Kyseisille poikkeuksille on toteutettu vastaavat poikkeuskäsittelijät ytimen ohjelmakoodin puolelle, jotka on toteutettava taulukon 2 mukaisella tavalla.

Taulukko 2. Järjestelmäpoikkeusten käsittelijät FreeRTOS-käyttöjärjestelmässä.

Järjestelmäpoikkeus	Käyttöjärjestelmän poikkeuskäsittelijä
SVCall	vPortSVCHandler
PendSV	xPortPendSVHandler
SysTick	xPortSysTickHandler

Reaaliaikakäyttöjärjestelmän vuorottaja vaati toimiakseen mekanismin, jonka avulla voidaan tarkistaa säännöllisin väliajoin, että onko jokin säie saavuttanut määräajan tai onko tietty synkronointipiste vapautettava. Sitä varten hyödynnetään prosessorin järjestelmäajastinta, joka tuottaa SysTick-järjestelmäpoikkeuksen säännöllisin väliajoin. Kuten taulukosta 2 ilmenee, SysTick-järjestelmäpoikkeus käsitellään ytimen toimesta xPortSysTickHandler-funktiossa, joka taas kutsuu ytimen xTaskIncrementTick-funktiota (kuva 6).

```
void xPortSysTickHandler( void )
{
    /* The SysTick runs at the lowest interrupt priority, so when this interrupt
    executes all interrupts must be unmasked.  There is therefore no need to
    save and then restore the interrupt mask value as its value is already
    known. */
    portDISABLE_INTERRUPTS();
    {
        /* Increment the RTOS tick. */
        if( xTaskIncrementTick() != pdFALSE )
        {
            /* A context switch is required.  Context switching is performed in
            the PendSV interrupt.  Pend the PendSV interrupt. */
            portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
        }
    }
    portENABLE_INTERRUPTS();
}
```

Kuva 6. Kuvakaappaus FreeRTOS-käyttöjärjestelmän lähdekoodin xPortSysTickHandler-funktiosta, joka käsittelee SysTick-järjestelmäpoikkeuksen.

Kyseisessä funktiossa tarkistetaan säikeiden määräaikojen toteutuminen. Mikäli käy ilmi, että säikeen suoritusvuoro on vaihdettava, kutsutaan viivästettyä ohjelmakeskeytystä (PendSV).

Säikeen suoritusvuoronvaihtoa ei voida tehdä saman tien, vaan ensiksi täytyy käsitellä matalamman prioriteetin poikkeukset, mikäli niitä on jonossa. Tästä syystä käytetään viivästettyä ohjelmakeskeytyskutsua, josta aiheutuva poikkeus käsitellään vasta, kun kaikki korkeamman prioriteetin poikkeukset ovat käsitelty.

Viivästetyn ohjelmakeskeytyskutsun poikkeuskäsittely suoritetaan ytimen toimesta xPortPendSVHandler-nimisessä funktiossa, jonka tehtävänä on suorittaa säikeen suoritusvuoron- ja kontekstinvaihto. Yksinkertaisesti selostettuna kyseinen funktio kopioi prosessorin rekistereiden arvot säikeen kontrollilohkoon, etsii säielistasta korkeimman prioriteetin säikeen, joka on valmiina ohjelmansuoritusta varten ja palauttaa kyseisen säikeen kontrollilohkoon tallennetut vuoronvaihtoa edeltäneet prosessorin rekistereiden arvot.

3.7 Semaforit

Semafori (engl. semaphore) on käyttöjärjestelmän ytimen hallinnoima komponentti, jota käytetään säikeiden synkronointiin ja resurssien suojaukseen tapauksissa, joissa resurssia voidaan käyttää useamman säikeen toimesta yhtäaikaisesti.

Riippuen käyttöjärjestelmästä, semaforeja on kahdenlaisia: binäärisemaforeja (engl. binary semaphore) ja laskevia semaforeja (engl. counting semaphore). Jälkimmäinen on tarkoitettu määrällisesti rajallisen resurssin suojaamiseen ja binäärisemaforit säikeiden synkronointiin ja kommunikointiin. Lähtökohtaisesti binäärisemaforia ei tulisi käyttää resurssien suojaamiseen, vaan sitä varten on tarkoitettu poissulkukomponentti (engl. mutex). Binäärisemaforin ja poissulun toiminta on hyvin samankaltainen, mutta toiminnallisuus voi erota kahden merkittävän seikan perusteella, jotka selviävät seuraavassa aliluvussa.

Laskevien semaforien avulla voidaan rajoittaa resurssin käyttöä, jota on saatavilla rajallinen määrä, eli vaikkapa tiedostojärjestelmän avattujen tiedostojen määrää. Laskevaa semaforia voidaan ajatella myös kokonaislukumuuttujana, mutta ohjelmakoodia suoritettaessa säikeissä tilanne on ongelmallinen, sillä muuttujan arvon vertailu, jonka käyttö on jaettu useamman säikeen kesken, ei ole atominen operaatio.

Esimerkkikoodissa 4 on kuvitteellinen tilanne, jossa kaksi eri säiettä käyttävät rinnakkaisesti rajallista resurssia, jonka lukumäärästä pidetään kirjaa kokonaislukumuuttujan avulla. Esimerkkikoodin kaltainen toteutus voi toimia hyvinkin pitkään ilman ongelmia.

Ongelmat kuitenkin ilmaantuvat, mikäli käytetään irrottavaa vuorotusalgoritmia ja säikeen suoritusvuoronvaihto tapahtuu if-ehtolauseen toteutumisen jälkeen: juuri ennen käytettyjen resurssien lukumäärää kuvaavan kokonaislukumuuttujan arvon kasvatusta. Suoritusvuoron saanut rinnakkainen säie pääsee käyttämään kaikki rajalliset resurssit ja suoritusvuoro palautetaan keskeytetylle säikeelle. Aiemmin keskeytetty säie pääsee taas jatkamaan ohjelmansuoritusta samasta pisteestä, josta se alun perin keskeytettiin, eli juuri ennen kokonaislukumuuttujan kasvatusta ja rajallisen resurssin käyttämistä, mikä tulee johtamaan virhetilanteeseen.

Esimerkkikoodin kaltainen ohjelmakoodi ei korjaannu, vaikka perään lisättäisiin useampi ehtolause, sillä resurssien määrää kuvaavan kokonaislukumuuttujan arvo on mitä todennäköisemmin kopioitu prosessorin rekisteriin. Tällöin muutettaessa muuttujan arvoa rinnakkaisesta säikeestä, ei muutos tule välittymään toiselle säikeelle, ennen kuin muuttujan arvo luetaan uudelleen prosessorin rekisteriin.

Ongelma ei myöskään ratkea, mikäli käytetään volatile-avainsanaa kuvaamaan muuttujan tietotyyppiä, joka kertoo ohjelmakielikääntäjälle, että muuttujan arvo saattaa muuttua hetkenä minä hyvänsä ohjelman kulkuvirtaan nähden, eli asynkronisesti. Nimittäin ehtolauseen tarkistus ja muuttujan arvon kasvatus ei ole atominen operaatio, vaan siinä on useampi konekäsky välissä, jonka aikana säikeen suoritusvuoro voi vaihtua.

```
#define RESOURCE_COUNT 5

static int resources_used;

// Example thread 0 body.

static void thread_0(void)
{
    /* Start of the critical section. */

    if ( resources_used < RESOURCE_COUNT )
    {
         // At this point all available resources may have been
         // consumed if thread switching occurred after if-condition
         // checking and before this variable was updated.

         resources_used++;

         use_limited_resource();
    }

    /* End of the critical section. */
}

// Example thread 1 body.

static void thread_1(void)
{
    /* Start of the critical section. */

    if ( resources_used < RESOURCE_COUNT )
    {
         resources_used++;

         use_limited_resource();
    }

    /* End of the critical section. */
}
```

Esimerkkikoodi 4. C-ohjelmakoodi, joka kuvaa ongelmakohtia, mikäli ohjelmakoodin kriittisiä kohtia yritetään suojata kokonaislukumuuttujan avulla.

Esimerkkikoodin 4 kaltainen tilanne voidaan korjata käyttöjärjestelmän ytimen hallinnoiman komponentin avulla, esimerkiksi laskevan semaforin avulla. Esimerkkikoodi 5 havainnollistaa, miten esimerkkikoodin 4 kaltainen tilanne voidaan toteuttaa laskevan semaforin avulla FreeRTOS-käyttöjärjestelmässä.

```
#include <FreeRTOS.h>
#include <semphr.h>

#define RESOURCE_COUNT 5

static SemaphoreHandle_t resource_sem;

// Example thread 0 body.

static void thread_0(void)
{
    if ( xSemaphoreTake(resource_sem, portMAX_DELAY) == pdTRUE )
    {
         use_limited_resource();
    }
}

// Example thread 1 body.

static void thread_1(void)
{
    if ( xSemaphoreTake(resource_sem, portMAX_DELAY) == pdTRUE )
    {
         use_limited_resource();
    }
}

int main(void)
{
     BaseType_t rc;

     // Create counting semaphore with maximum value of RESOURCE_COUNT
     // and initial value of RESOURCE_COUNT.

     resource_sem= xSemaphoreCreateCounting(RESOURCE_COUNT, RESOURCE_COUNT);
     if ( !resource_sem )
         return(-1);

     // ...
}
```

Esimerkkikoodi 5. C-ohjelmakoodi, joka kuvaa laskevien semaforien käyttöä FreeRTOS-käyttöjärjestelmässä.

3.8 Poissulku

Useamman säikeen rinnakkaisaikaisen suorituksen yhteydessä törmätään useasti tilanteeseen, jossa kahden tai useamman säikeen on päästävä käsittelemään samaa resurssia. Ohjelmansuorituksen kohtaa, jossa tämänkaltaista resurssia käsitellään, kutsutaan kriittiseksi osaksi (engl. critical section). Kyseessä on ohjelmansuorituksen kannalta

vaarallinen tilanne, koska kyseisen resurssin eheys tai resurssia käsittelevän säikeen ohjelmansuoritus saattaa vaarantua [28, s. 75].

Kuvitellaan tilanne, jossa on kaksi eri prioriteetin omaavaa säiettä, jotka molemmat käsittelevät samaa merkkijonoa. Matalamman prioriteetin säie kirjoittaa dataa merkkijonoon silmukassa, kunnes ydin keskeyttää säikeen ohjelmansuorituksen ja antaa suoritusvuoron korkeamman prioriteetin omaavalle säikeelle. Kyseinen säie kirjoittaa myös dataa samaan merkkijonoon, jonka jälkeen suoritusvuoro palautetaan takaisin matalamman prioriteetin säikeelle. Jos kyseisen merkkijonon käsittely silmukassa on toteutettu edes jokseenkin järkevästi, niin ohjelmansuoritus ei tämän takia kaadu virhetilanteeseen, mutta kyseisen merkkijonon eheys on vaarantunut eikä merkkijonon data ole enää ohjelmansuorituksen kannalta kelvollinen.

Edellä kuvaillun tilanteen välttämäiseksi hyödynnetään keskinäistä poissulkua (engl. mutual exclusion). Keskinäinen poissulku toteutetaan käyttöjärjestelmän hallinnoiman poissulkukomponentin avulla. Kyseisen komponentin toiminta on hyvin samankaltainen binäärisemaforin kanssa, mutta käyttötarkoitus on täysin erilainen poissulun omistajuussuhteen takia.

Poissulkujen toteutus ja käyttö on täysin käyttöjärjestelmäkohtainen seikka, mutta yleinen toimintaperiaate on kaikissa sama: Ohjelmansuorituksen kriittisen osan alkupisteessä säie lukitsee poissulkuobjektin. Mikäli kyseinen poissulku on jo lukittu, lukitsemista yrittävä säie joutuu odottamaan sen vapautumista. Kriittisen osuuden jälkeen säie vapauttaa poissulun, jolloin rinnakkainen säie pääsee lukitsemaan poissulun.

Poissulkua ei voida kiinnittää mihinkään tiettyyn resurssiin, eikä näin ollen resurssin suojauksesta ole mitään taetta, vaan poissulun onnistunut toiminta riippuu ohjelmakoodin kirjoittajasta. Poissulkua voidaan ajatella eräänlaisena aitana, jonka yli pääsee vain yksi käyttäjä kerrallaan.

POSIX.1-2008-standardissa on määritelty poissulkujen toiminnallisuus, josta seuraa merkittävin ero poissulun ja binäärisemaforin välille: poissulkuobjekti voidaan vapauttaa vain ja ainoastaan säikeen toimesta, joka on sen lukinnut [6, 2.9.3]. Poissulkuobjektin omistajuussuhteen takia voi syntyä pysyvä lukko, jonka takia useampi säie ei pääse

enää jatkamaan ohjelmansuoritusta yrittäessään lukita poissulkuobjekteja, jotka ovat lukittu toisistaan riippuvien säikeiden toimesta. Kyseisestä tilanteesta käytetään nimitystä kuolemanlukko (engl. death lock).

Pysyvien lukkotilanteiden välttämiseksi POSIX.1-2008-standardissa on määritelty reaaliaikakäyttöjärjestelmien osalta priorisointimenettely poissulkuobjektin lukitsemisen suhteen: jos korkeamman prioriteetin säie pyrkii lukitsemaan poissulun, joka on lukittu matalamman prioriteetin säikeen toimesta, tulee kyseinen matalamman prioriteetin säikeen periä lukitsemista pyrkivän säikeen prioriteetti (engl. priority inheritance).

FreeRTOS-käyttöjärjestelmä toteuttaa poissulkujen prioriteetin perinnän oletusarvoisesti ja se on FreeRTOS-käyttöjärjestelmän ainoa ero binäärisemaforeihin verrattuna [7, s. 251]. POSIX-rajapintoihin perustuvien reaaliaikakäyttöjärjestelmien osalta kyseinen prioriteetin perintä täytyy määritellä erikseen poissulkukomponentin attribuuttirajapintojen kautta.

3.9 Muistinhallinta FreeRTOS-käyttöjärjestelmässä

Käytettäessä C-standardikirjastoa, dynaamiseen muistinvaraukseen liittyvät funktiot eivät sellaisenaan sovellu käytettäväksi sulautettujen järjestelmien ohjelmistoissa. Mahdolliset ongelmat liittyvät säieturvallisuuteen (engl. thread-safety), muistialueen pirstoutumiseen (engl. fragmentation) sekä varauskutsujen deterministisyyteen reaaliaikasovelluksissa. [27, s. 124.]

POSIX.1-2008-standardi vaatii, että C-standardikirjaston mukaiset muistinvaraus ja -vapautus funktiot ovat säieturvallisia [6, B.2.9.1], mutta näin ei kuitenkaan oletusarvoisesti ole käytettäessä C-standardikirjaston riisuttuja versioita, kuten newlib- ja newlib-nanostandardikirjastoja, jotka vaativat erillisten poissulkujen toteutuksen varausfunktion yhteyteen [29].

FreeRTOS-käyttöjärjestelmä tarjoaa vaihtoehtoiset menetelmät kekomuistin (engl. heap memory) varaamiseen pvPortMalloc- ja vPortFree-funktioiden avulla, joista on FreeRTOS 10.0 -versioissa viisi eritasoista algoritmia. Kaikki algoritmit ovat säieturvallisia, ja

heap_1-, heap_2- tai heap_4-algoritmit varaavat muistilohkot staattisesti varatusta muistitaulukosta, jonka koko on määritelty ohjelman käännösvaiheessa.

heap_1

heap_1 algoritmi sallii muistin varaamisen, mutta ei vapauttamista. Kyseessä on täysin deterministinen algoritmi ja näin ollen myös soveltuu reaaliaikasovelluksiin [7, s. 29]. Turvallisuus perustuu siihen, että käyttöjärjestelmän ydin varaa kaiken säikeiden kontrollilohkoihin ja objekteihin tarvittavan muistin ennen vuorottajan käynnistymistä. Muisti varataan ennalta vain ja ainoastaan ytimen hallinnoimia objekteja varten, kuten semaforeja ja poissulkuja varten, joten algoritmin käyttö ei takaa turvallisuutta käyttöjärjestelmän ytimen ulkopuolella.

heap_2

Toisin kuin heap_1, heap_2 algoritmi sallii myös muistin vapauttamisen, mutta ei yhdistele vapautettuja lohkoja muistitaulukon vierekkäisiin vapaisiin lohkoihin. Tämä johtaa nopeasti muistitaulukon pirstaloitumiseen, mikäli ohjelman dynaamisesti varaamien lohkojen koko ei ole ennakoitavissa. Kyseinen algoritmi on vanhentunut FreeRTOS 9.0 -versiosta lähtien ja sen käyttö ei ole enää suositeltavaa. [7, s. 30.]

heap_3

heap_3 ei ole varsinainen muistinvarausalgoritmi, vaan ennemminkin poissulku C-standardikirjaston muistin varaus- ja vapautusfunktioita varten, tehden niistä säieturvallisia. Toiminta perustuu siihen, että muistinvaraus ja –vapautuskutsujen yhteydessä pysäytetään kaikkien säikeiden ohjelmansuoritus, kunnes muistivaraus on saatettu loppuun. [7, s. 32.]

heap_4

Kuten heap_1- ja heap_2-algoritmin kohdalla, heap_4-algoritmissa kekomuistitaulukko varataan staattisesti. Muistilohkoista pidetään yllä linkkilistaa ja varaus tapahtuu ensimmäisen sopivankokoisen lohkon löydyttyä. Toisinkuin heap_2-algoritmi, muistitaulukon

vierekkäiset vapaat lohkot yhdistellään yhdeksi vapaaksi lohkoksi. Tämä ei kuitenkaan estä muistitaulukon pirstaloitumista, mutta soveltuu sovelluksiin, joissa dynaamiseen muistinvaraukseen vaadittavien lohkojen kokoja ei voida ennalta määritellä. [7, s. 32.]

Koska vapaan muistilohkon etsimiseen kulunut aika kasvaa lineaarisesti suhteessa varattujen muistilohkojen määrään, ei kyseinen algoritmi ole deterministinen, eikä näin ollen sovellu turvakriittisiin tai reaaliaikasovelluksiin. Kyseinen algoritmi on kuitenkin säieturvallinen ja tehokkaampi, kuin C-standardikirjaston toteutus ja näin ollen soveltuu hyvin pehmeää reaaliaikaisuutta vaativiin sovelluksiin.

heap_5

heap_5 algoritmi perustuu heap_4-algoritmiin, mutta kekomuistin taulukko ei ole varattu staattisesti yhdestä jatkuvasta muistijonosta. Kyseinen algoritmi voi varata muistia useammasta eri ennalta määritetystä muistialueesta. Esimerkkikoodi 6 havainnollistaa kekomuistialueiden määrittelyn vuorottajan käynnistyksen yhteydessä. Muistitaulukon osoitteiden on oltava järjestetty pienimmästä osoitteesta suurimpaan ja taulukko täytyy päättää muistialueella, jonka koko on nolla. [7, s. 35.]

```
#include <inttypes.h>
#include <FreeRTOS.h>

#define DECLARE_HEAP_REGION(START_ADDRESS, SIZE) \
    {(uint8_t*)(START_ADDRESS), (SIZE)}

// Application heap memory regions array.
// Array addresses must be in address order, from low to high.
// Array must be terminated using zero sized region.

static const HeapRegion_t heap_memory[]=
{
     DECLARE_HEAP_REGION(0x200000000UL, 50 * 1024),
     DECLARE_HEAP_REGION(0x600000000UL, 10 * 1024),
     DECLARE_HEAP_REGION(0x900000000UL, 80 * 1024),
     DECLARE_HEAP_REGION(0, 0)
};

int main(void)
{
     // Initialize heap memory.

     vPortDefineHeapRegions(heap_memory);

     // Start scheduler.

     vTaskStartScheduler();

     return(0);
}
```

Esimerkkikoodi 6. Kekomuistin määrittely FreeRTOS-käyttöjärjestelmässä.

4 Tiedonkeräimen suunnittelu ja valmistus

4.1 Piirilevyn komponentit

Tiedonkeräinlaite suunniteltiin STMicroelectronicsin valmistaman STM32L4S5VIT6-mikrokontrollerin ympärille, joka toteuttaa 32-bittisen ARM Cortex-M4F -prosessorin.

Laite oli alun perin tarkoitus suunnitella Cortex-M3-prosessoriin pohjautuvan mikrokontrollerin ympärille, sillä laitteiston ohjelmistossa ei ole raskasta liukulukulaskentaa tai digitaalista signaalinkäsittelyä, joissa Cortex-M4F-prosessorin FPU- ja DSP-apuprosessoreista olisi hyötyä.

Mikrokontrollerin valintaan vaikutti enimmäkseen kyseisen mikrokontrollerin hyvä saatavuus, matala virrankulutus, RAM- ja FLASH-muistin suuri määrä sekä integroidut oheisjärjestelmäkomponentit, kuten useampi UART-, SPI- ja I^2C-ohjain.

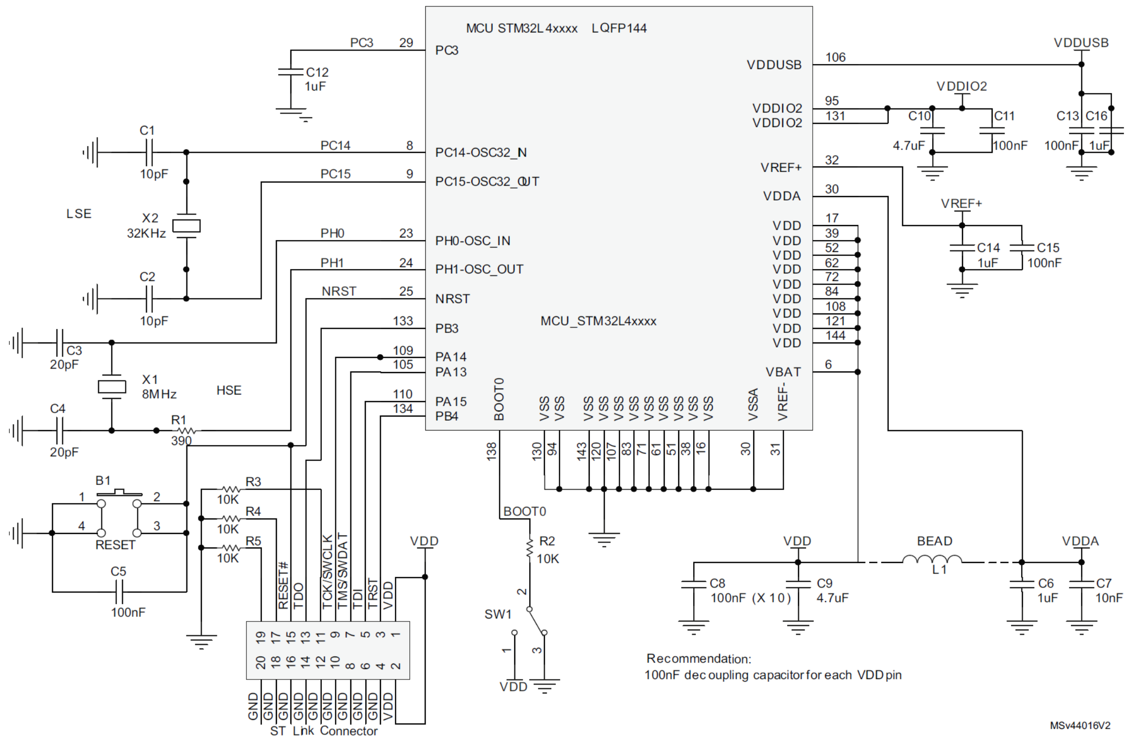
Lisäksi piirilevyn komponentteihin valikoitui USB-protokollan toteuttava FTDI FT230X -mikropiiri, sekä etäluentaa varten Wi-Fi- ja Bluetooth-moduulit.

4.2 Piirikaavion suunnittelu

Piirilevy suunniteltiin Eagle CAD -nimisen suunnitteluohjelman ilmaisversion avulla, jonka toiminnallisuuteen ja käyttöehtoihin liittyy tiettyjä rajoituksia. Ohjelman ilmaisversiolla tuotettuja suunnitelmia ei voi soveltaa kaupalliseen käyttöön ja piirilevylayoutin kerrosten lukumäärä on rajoitettu kahteen kerrokseen. Lisäksi yksittäisen projektin piirikaavioiden lukumäärä on myös rajoitettu kahteen kappaleeseen.

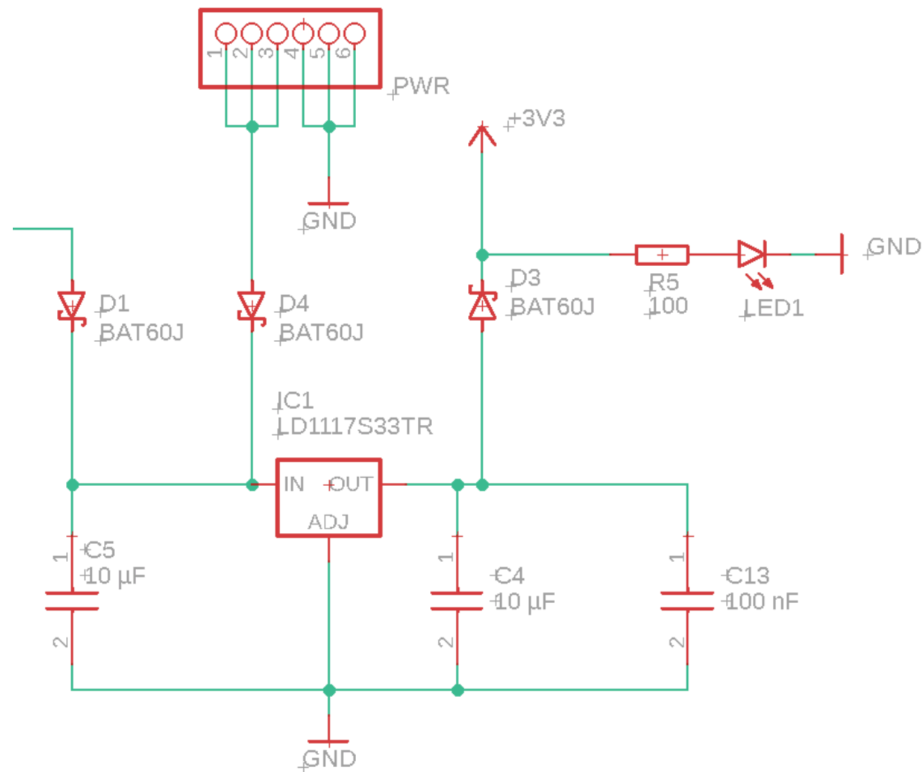
Piirilevyn piirikaavion suunnittelu tapahtui laitteeseen valittujen komponenttien datalehdessä esiteltyjen vähimmäisvaatimusten perusteella. Useimmat mikropiirivalmistajat tarjoavat myös tuotteilleen esimerkkisuunnitelmia, joita voi hyödyntää tuotteen suunnitteluvaiheessa. Tähän työhön valitun STM32L4S5VIT6-mikrokontrollerin esimerkkisuunnitelma on esitelty mikrokontrollerin resurssisuunnitelmalehtiössä (kuva 7), jonka lisäksi laitteeseen valitun mikrokontrollerin valmistajalta löytyy samaan mikrokontrollerisarjaan

pohjautuvia evaluointialustoja, joiden suunnitelmia voi myös hyödyntää suunnitteluvaiheessa.



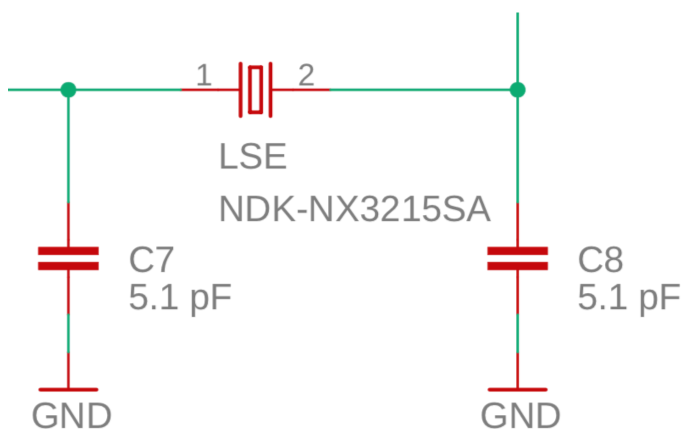
Kuva 7. STM32L4-mikrokontrollerin piirin referenssisuunnitelma [17, s. 49].

Piirikaavion suunnittelun ensimmäinen vaihe oli mikrokontrollerin virransyötön toteutus. Piirilevyn pääasiallinen tehonsyöttö tapahtuu FTDI-piiriin ja USB Micro-B 2.0 -liittimen kautta, jonka syöttöjännite on USB 2.0 -spesifikaation mukaan 5 V ± 5 % [15, s. 178]. Mikrokontrollerin datalehdestä käy ilmi, että mikrokontrollerin käyttöjännite V_{DD} tulisi olla vähintään 1,71 V ja korkeintaan 3,6 V [16, s 22]. Tämän lisäksi virransyötön rinnalla tulee olla vähintään yksi erotuskondensaattori, jonka kapasitanssi on vähintään 4,7 µF ja jokaista mikrokontrollerin V_{DD} -käyttöjännitetuloa kohden tulisi olla 100 µF:n erotuskondensaattori [17, s. 17]. Näiden tietojen pohjalta mikrokontrollerin virransyöttö toteutettiin LDO-regulaattorin avulla, jonka toteutus on esitelty kuvassa 8.



Kuva 8. Virransyötön toteutus.

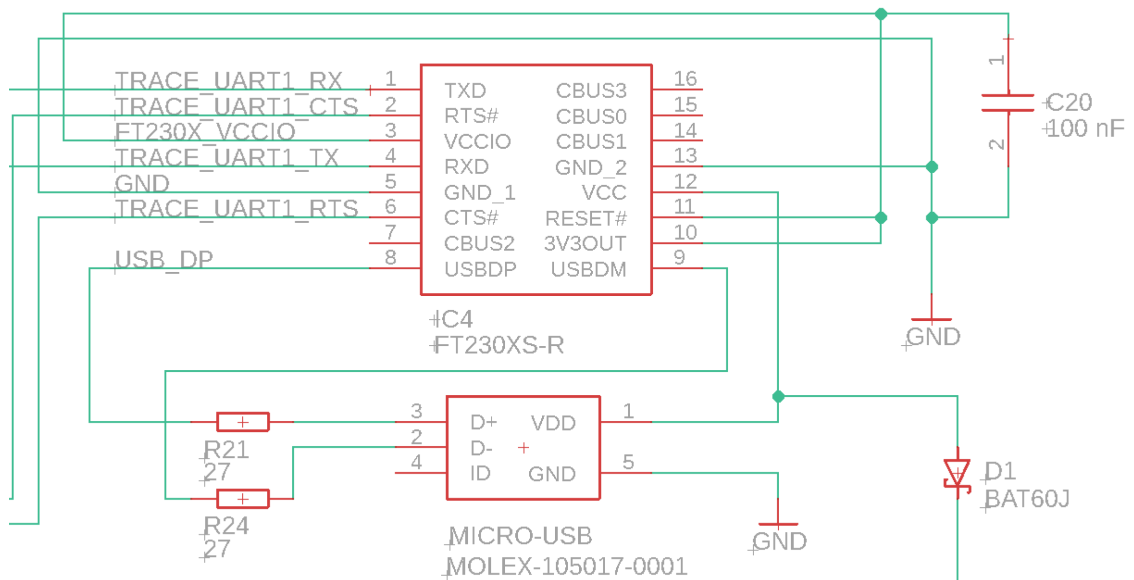
Laitteen mikrokontrolleri on määritelty käyttämään järjestelmäkellon lähteenä sisäistä 4 MHz:n oskillaattoria, mutta mikrokontrollerin STOP-virransäästötilan toteuttamista varten mikrokontrolleri vaati ulkoisen 32,768 kHz:n oskillaattorin [18, s. 5], jonka toteutus on esitelty kuvassa 9.



Kuva 9. LSE-oskillaattorin toteutus.

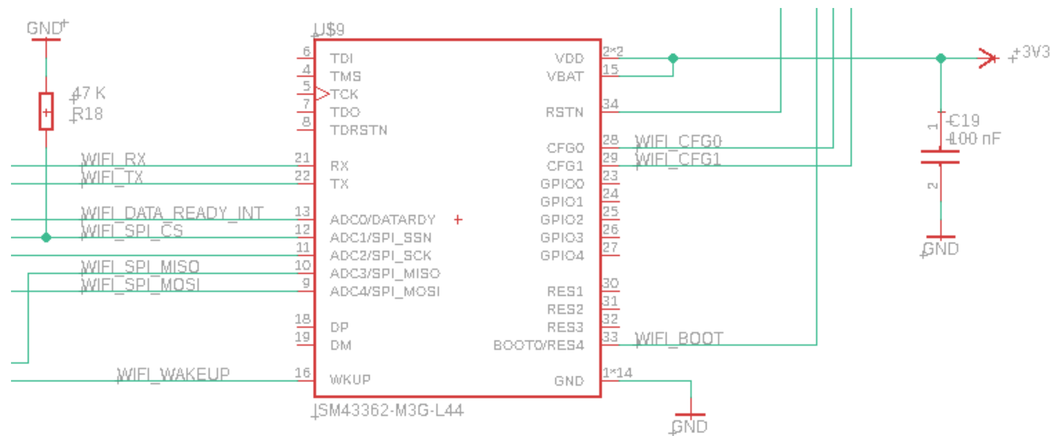
Mikrokontrollerin rinnalle lisättiin FTDI FT230X -mikropiiri, joka toteuttaa laitteen USB-protokollan konsolikäyttöliittymää varten (kuva 10). FT230X-mikropiiriltä tulevan datansiirto mikrokontrollerille on toteutettu asynkronisesti UART-ohjaimen kautta.

UART-ohjaimen vaadittavien TX- ja RX-signaalien lisäksi on reititetty Ready to Send (RTS) ja Clear to Send (CTS) -signaalit, joiden avulla halitaan tietovirtaa (engl. flow control). Vaikka mikrokontrollerin ohjelma on toteutettu siten, että dataa luetaan DMA-ohjaimen avulla rengaspuskuriin (engl. ring buffer), mutta koska data käsitellään asynkronisesti, voi tulla tilanteita, joissa dataa ei voida käsitellä tarpeeksi nopeasti, ja tuleva data korvaa rengaspuskurin käsittelemättömän datan. RTS- ja CTS-signaalien avulla ilmaistaan, milloin dataa voidaan vastaanottaa ja lähettää, mikä ehkäisee rengaspuskurin datan ylityksen.



Kuva 10. FT230X-mikropiirin toteutus.

Wi-Fi-moduulin ja mikrokontrollerin välistä kommunikointia varten on reititetty UART-ohjaimen ja SPI-väylän signaalit (kuva 11), tosin molempien kommunikointimenetelmien samanaikainen käyttö ei onnistu Wi-Fi-moduulin sisäisen ohjelmiston rajoitusten vuoksi.

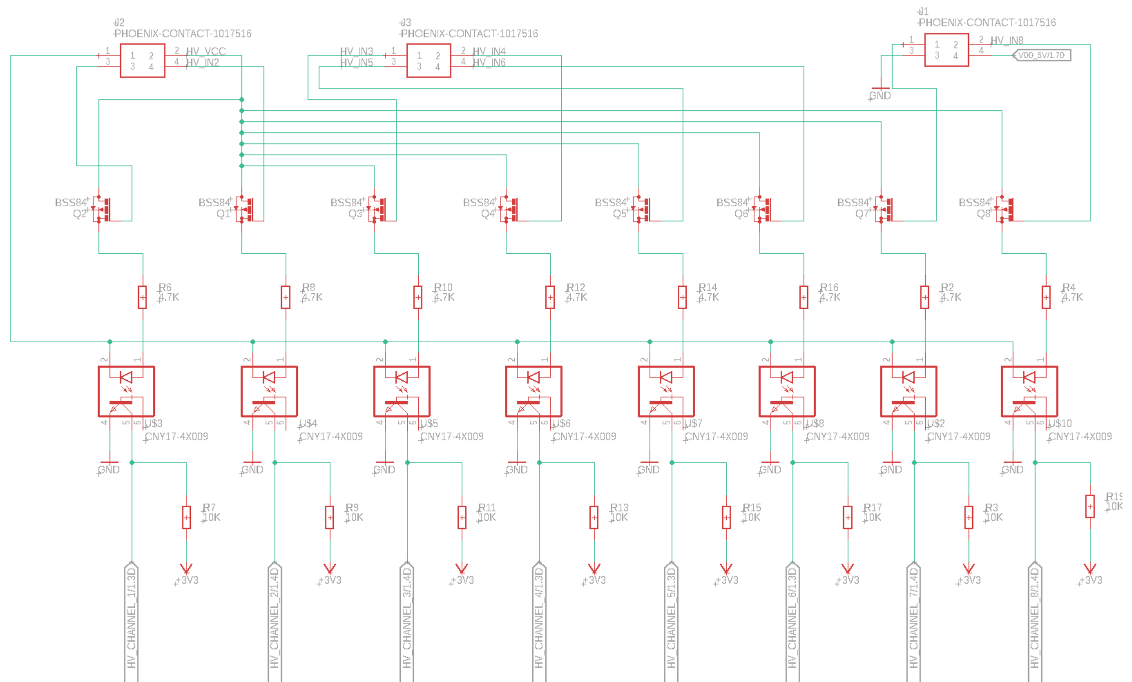


Kuva 11. Wi-Fi-moduulin toteutus.

Mikrokontrollerin ohjelmointi ja virheenjäljitys tapahtuu Serial Wire Debug (SWD) -protokollan kautta, joka toteutetaan kahden I/O-signaalin avulla. Laitteen SWD-porttiin reititettiin lisäksi virheviestienjäljityssignaali (Serial Wire Out, SWO), jota käytetään laitteen toissijaisena virheenjäljitysviestien välitysmenetelmänä.

Laitteen 24 V:n tulojännitesignaalien lukeminen on toteutettu optoisolaattorien avulla (kuva 12). Laitteeseen valittujen optoisolaattorien toiminta perustuu infrapunahohtodiodiin: infrapunahohtodiodi ohjaa optoisolaattorin sisäistä NPN-tyyppistä fototransistoria, joka muodostaa galvaanisen erotuksen mikrokontrollerin käyttöjännitteen ja 24 V:n tulojännitesignaalin välille.

Optoisolaattorien kytkentä on toteutettu siten, että sisäinen NPN-transistori ohjaa ulkoisen ylösvetovastuksen kautta laitteen 3,3 V:n käyttöjännitettä mikrokontrollerin GPIO-porttiin, joka saa aikaan sen, että GPIO-portin 3,3 V:n signaali putoaa nollaan (looginen nolla), mikäli optoisolaattorin kautta kulkee 24 V:n tulosignaali.



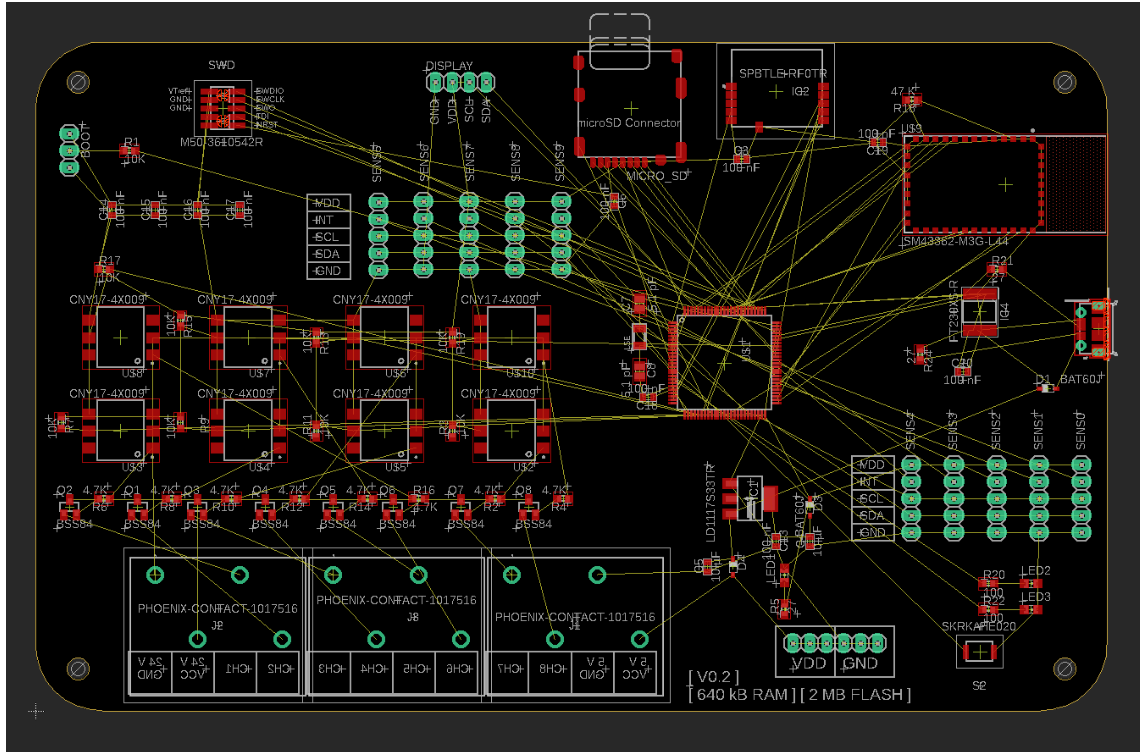
Kuva 12. 24 V DC tulojen reititys optoisolaattorien kautta.

4.3 Piirilevyn layoutin ja komponenttien reititys

Piirilevyn layoutilla tarkoitetaan piirilevyn kuviota, joka muodostaa piirilevyn ulkomuodon ja komponenttisymbolien reitityksen. Tämän työn piirilevy koostuu kahdesta kerroksesta: päällimmäinen kerros sisältää kaikki pintaliitoskomponentit ja alimmainen kerros muodostaa maatason (engl. ground plane).

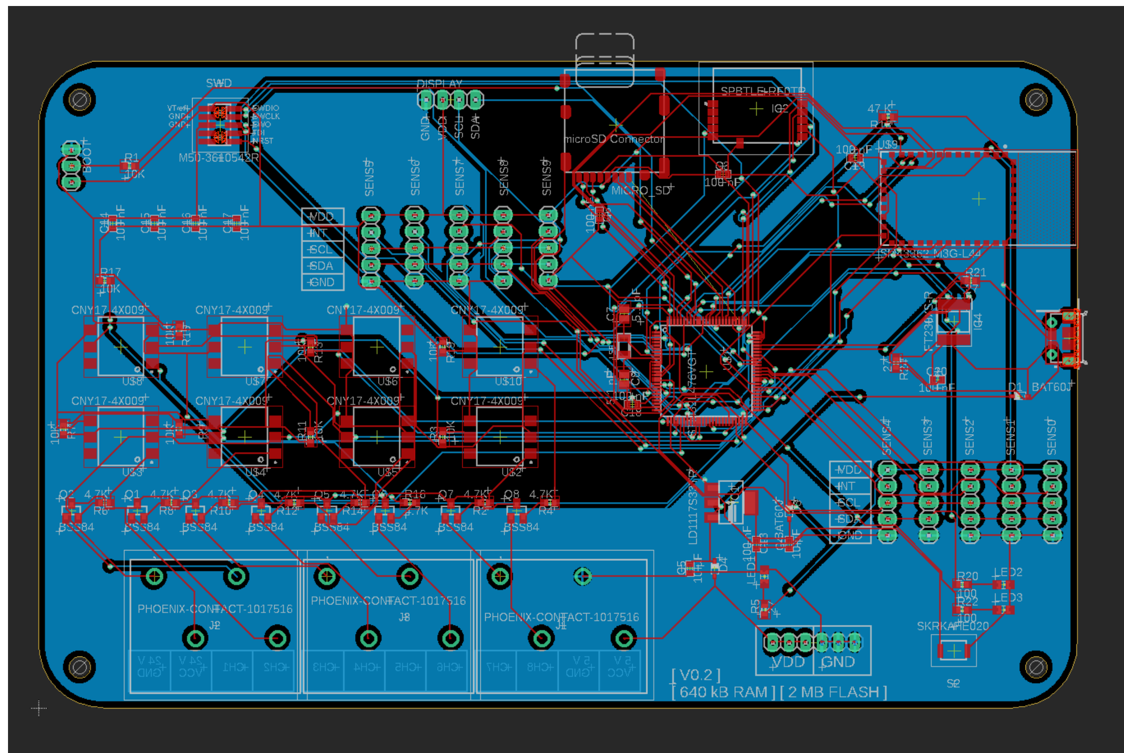
Piirilevyn layoutin toteutus alkoi komponenttien sijoittelusta: ensiksi sijoitettiin komponentit, jotka tuli olla piirilevyn ulkoreunassa, kuten 24 V:n terminaaliliittimet, µSD-muistikorttipaikka sekä USB-liitin. Lisäksi ulkoreunalle sijoitettiin Wi-Fi- ja Bluetooth-moduulit, sekä SWD-portti, jotka kaikki yhdessä muodostivat piirilevyn ulkomuodon ja mitat. Komponenttien sijoittelu eteni optoisolaattoreista, sekä 24 V:n tulosignaalipiirin MOSFET:eista, jotka pyrittiin sijoittamaan mahdollisimman symmetrisesti terminaaliliittimien läheisyyteen.

Kun suurimmat komponentit olivat pois tieltä, sijoitettiin mikrokontrolleri ja sen kriittiset komponentit, kuten oskillaattori ja suodatuskondensaattorit, jotka tuli sijoittaa mahdollisimman lähelle mikrokontrolleria pitkien vetojen välttämiseksi. Kuvassa 13 on esitelty piirikuvion komponenttien reititystä edeltänyt tilanne.



Kuva 13. Layout komponenttien sijoittelun jälkeen.

Komponenttien sijoittelun jälkeen layoutin toteutus eteni komponenttien reitityksellä, joka suoritettiin käyttäen suunnitteluohjelman automaattista reititystoimintoa. Reititys viimeisteltiin toteuttamalla piirilevyn kääntöpuolelle maakerros, joka näkyy kuvassa 14 sinisellä värillä.

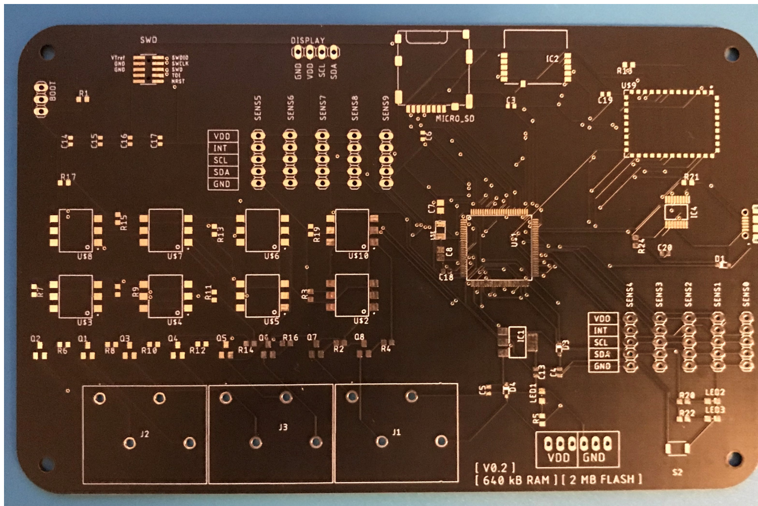


Kuva 14. Piirikuvio komponenttien reitityksen jälkeen.

4.4 Piirilevyn Gerber-tiedostojen luonti ja valmistus

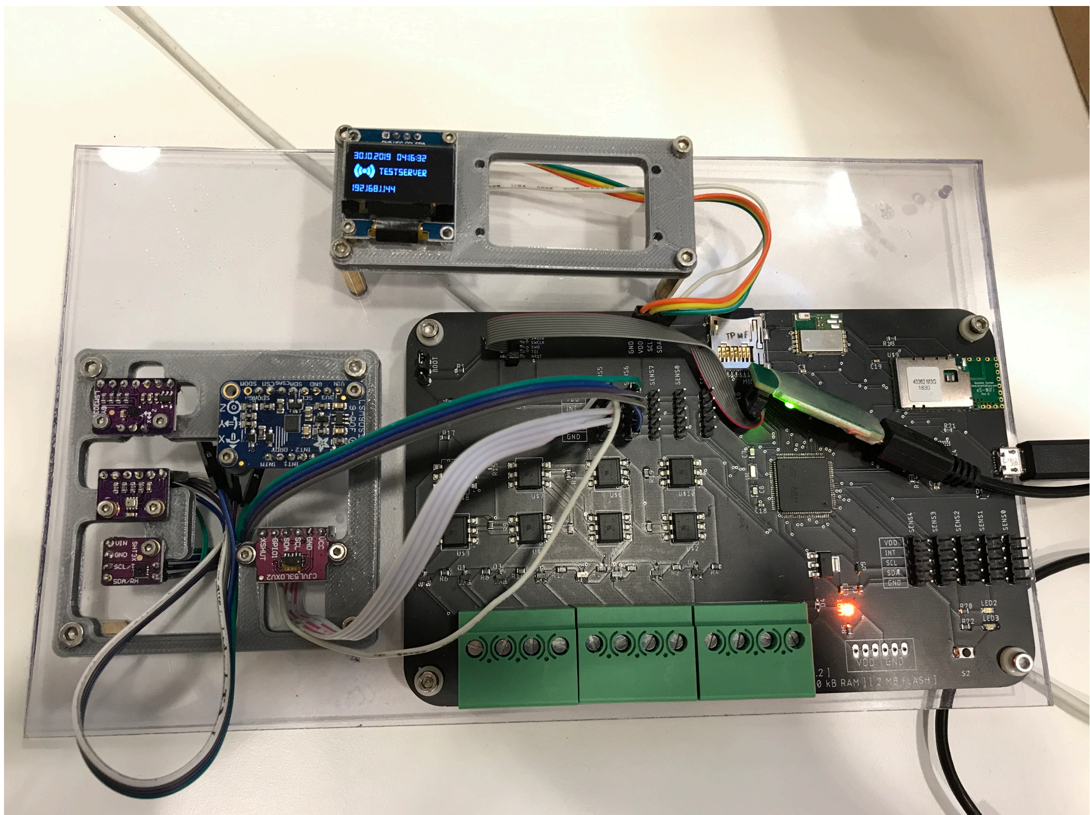
Gerber-tiedostoilla tarkoitetaan piirikuviosta muodostettuja porakuviotiedostoja, joiden perusteella piirilevy tuotetaan piirilevyvalmistajan toimesta. Nimityksen mukaisesti tiedostot ovat Gerber-tiedostoformaatissa, joka on vakiintunut tiedostomuoto piirilevyteollisuudessa [23, s. 430].

Piirilevysuunnittelun viimeisessä vaiheessa kehitettiin Gerber-tiedostot, joiden avulla valmistettiin viiden kappaleen prototyyppierä ulkomaisen piirilevyvalmistajan toimesta (kuva 15).



Kuva 15. Piirilevy ennen komponenttien ladontaa.

Laitteiston komponentit ladottiin ja juotettiin käsin, minkä lopputulos ilmenee kuvassa 16.



Kuva 16. Valmis laite.

5 Ohjelmiston toteutus

5.1 Ohjelmointiympäristö

Projektin ohjelmointiympäristö toteutettiin GNU Make -työkalun avulla. GNU Make on tiedostokäsittelyn automatisointiin tarkoitettu työkalu, joka on laajasti käytetty Unix-pohjaisissa järjestelmissä avoimen lähdekoodin ohjelmien kääntämistä varten. GNU Make -työkalun toiminta perustuu Makefile-tiedostoon, johon määritellään sääntöjä, miten minkäkin tyyppistä tiedostoa tulisi käsitellä.

GNU Make -työkalun toiminta ei ole kuitenkaan rajattu mihinkään tiettyyn ohjelmointikieleen tai tiedostoformaattiin, jonka takia GNU Make -työkalulla voi tehdä paljon muutakin kuin tiedostojen käsittelyä. Esimerkiksi tässä projektissa esikäännösvaiheessa suoritetaan ohjelma, joka muuttaa käyttöliittymässä käytettävät fontti- ja bittikarttatiedostot binäärimuotoon.

Ohjelman lähdekoodin kääntäminen ja linkitys tapahtuu ARM EABI -binäärirajapintaa varten tarkoitettujen GNU-käännöstyökalujen (GCC, engl. GNU Compiler Collection) avulla, jotka sisältävät myös GNU-virheenjäljitysohjelman (GDB, engl. GNU Debugger). Näitä kaikkia työkaluja käytetään shell-skriptin avulla, jolle annetaan kaksi parametria. Ensimmäinen parametri kertoo toteutettavan komennon, jotka ovat ohjelman kääntäminen (make), virheenjäljitys (debug) tai ohjelman lataus laitteeseen (flash). Toinen komento kertoo kohdelaitteen, jota varten komento suoritetaan. Projektin kohdelaitteet ovat Unix-pohjainen simulaattori sekä itse projektissa toteutettu STM32L4-mikrokontrolleripohjainen laite.

Ohjelman lataaminen laitteeseen sekä ohjelman virheenjäljitys tapahtuu J-Link EDU-mini -ohjelmointityökalun avulla, jota käytetään myös GDB-serverinä GNU Debugger -virheenjäljitysohjelman yhteydessä.

5.2 Rakenne

Ohjelmisto kehitettiin sillä periaatteella, että sitä voidaan suorittaa käyttöjärjestelmäriippumattomassa ympäristössä. Tarkoituksena oli, että samaa ohjelmaa voidaan suorittaa eri kohdelaitteissa, jotka ovat tämän projektin tapauksessa mikrokontrolleri sekä Unix-pohjainen tietokone (simulaattori).

Tätä varten ohjelmaan kehitettiin ylimääräinen abstraktiokerros (taulukko 3), jonka kautta käsitellään kohdelaiteriippuvaisia aliohjelmakutsuja - varsinainen sovelluskerros näyttää samalta eri kohdelaitteiden välillä. Esimerkiksi uuden säikeen luominen tapahtuu kutsumalla sysCreateThread-nimistä funktiota. Riippuen kohdekäyttöjärjestelmästä kyseinen funktio luo säikeen käyttäen POSIX- tai FreeRTOS-rajapintaa.

Taulukko 3. Esimerkki abstraktiokerroksen käyttöjärjestelmäriippumattomista funktioista.

Funktio	**Selite**
sysCreateThread	Säikeen luonti.
sysSignalThread	Säikeen signalointi.
sysSignalWait	Säikeen sisäinen signaalin odotusfunktio.
sysKillCurrentThread	Säikeen ohjelmasuorituksen lopetus.
sysCreateMutex	Poissulun luonti.
sysLockMutex	Poissulun lukitseminen (asynkroninen odotus).
sysTryLockMutex	Poissulun lukitsemisyritys, palaa saman tien.
sysUnlockMutex	Poissulun avaus.
sysCreateSemaphore	Semaforin luonti.
sysLockSemaphore	Semaforin lukitseminen.
sysTryLockSemaphore	Semaforin lukitsemisyritys, palaa saman tien.
sysUnlockSemaphore	Semaforin avaus.
sysStartScheduler	Ytimen vuorottajan käynnistys.
sysShellRun	Konsolin suoritus (asynkroninen odotus).
sysIsInISR	Funktio, jonka avulla tarkistetaan, että suoritetaanko poikkeuskäsittelyä
sysPlatformInit	Laitteen oheisjärjestelmäkomponenttien alustus.

5.3 Käynnistys

Cortex-M4-prosessorin käynnistyessä ohjelmansuoritus alkaa aina vektoritaulukosta. Kuten teoriaosuudessa jo mainittiin, vektoritaulukon ensimmäinen alkio on pinomuisti alkuosoite, jota seuraa poikkeusvektorit (esimerkkikoodi 7).

```
// ISR function type.

typedef void (*const SYS_ISR_FUNCTION)(void);

// Vector table type.

typedef struct
{
    uint32_t*               stack_ptr;
    const SYS_ISR_FUNCTION isr_handlers[SYS_MAX_VECTOR_COUNT];
}
SYS_VECTOR_TABLE;

// Vector table.

SYS_USED_SECTION_NAME(".isr_vector")
const SYS_VECTOR_TABLE stm32l4_custom_v02_vector_table =
{
    // Initial stack pointer.

    (uint32_t*)&sys_stack_start,
    {
        // Cortex-M4 vectors.

        SYS_ISR_HANDLER(Reset_Handler),
        SYS_ISR_HANDLER(NMI_Handler),
        SYS_ISR_HANDLER(HardFault_Handler),
        SYS_ISR_HANDLER(MemManage_Handler),
        SYS_ISR_HANDLER(BusFault_Handler),
        SYS_ISR_HANDLER(UsageFault_Handler),
        0,
        0,
        0,
        0,
        SYS_ISR_HANDLER(SVC_Handler),
        SYS_ISR_HANDLER(DebugMon_Handler),
        0,
        SYS_ISR_HANDLER(PendSV_Handler),
        SYS_ISR_HANDLER(SysTick_Handler),

        // Vendor specific vectors.

        // ...
    }
};
```

Esimerkkikoodi 7. Otanta ohjelman vektoritaulukosta, jossa sisältää Cortex-M4-prosessorin poikkeuskäsittelijät. Vektoritaulukon pituuden vuoksi esimerkistä on jätetty oheisjärjestelmäkomponenttien poikkeusvektorit, joita on yhteensä 128.

Jotta vektoritaulukko saadaan oikeaan muistiosoitteeseen, määritellään vektoritaulukolle muistialueattribuutti (.isr_vector), joka ohjelman linkityksen yhteydessä sijoitetaan FLASH-muistin alkupäähän linkkeri-skriptin avulla (esimerkkikoodi 8).

```
MEMORY
{
    RAM          (xrw)  : ORIGIN = 0x20000000, LENGTH = 640K
    VECTOR_TABLE (rx)   : ORIGIN = 0x08000000, LENGTH = 570
    FLASH        (rx)   : ORIGIN = 0x08000240, LENGTH = 2047K
}

sys_stack_start =  ORIGIN(RAM) + LENGTH(RAM);

SECTIONS
{
    .isr_vector :
    {
      . = ALIGN(8);
      KEEP(*(.isr_vector))
      . = ALIGN(8);
    } > VECTOR_TABLE

    /* ... */
}
```

Esimerkkikoodi 8. Vektoritaulukon sijoitus FLASH-muistin alkuun linkkeri-skriptin avulla.

Käynnistyksen yhteydessä prosessori siirtyy käsittelemään Reset-poikkeusta, joka käsitellään vektoritaulukon osoittamassa Reset_Handler-funktiossa (esimerkkikoodi 9). Reset_Handle-funktio suorittaa

- staattisesti varattujen muuttujien muistialueen nollauksen
- prosessorin kello-oskillaattorin alustuksen SystemInit-funktion kautta
- main-funktion kutsun, eli pääohjelmaan siirtymisen.

```
void Reset_Handler(void)
{
    __asm volatile
    (
    " @ set stack pointer                  \n"
    " ldr   sp, =sys_stack_start           \n"
    " movs  r1, #0                         \n"
    " b     LoopCopyDataInit               \n"
    " CopyDataInit:                        \n"
    "     ldr    r3, =_sidata              \n"
    "     ldr    r3, [r3, r1]              \n"
    "     str    r3, [r0, r1]              \n"
    "     adds   r1, r1, #4                \n"
    " @ Copy the data segment initializers \n"
    " @ from flash to SRAM                 \n"
    " LoopCopyDataInit:                    \n"
    "     ldr    r0, =_sdata               \n"
    "     ldr    r3, =_edata               \n"
    "     adds   r2, r0, r1                \n"
    "     cmp    r2, r3                    \n"
    "     bcc    CopyDataInit              \n"
    "     ldr    r2, =_sbss                \n"
    "     b      LoopFillZerobss           \n"
    "                                      \n"
    " @ Zero fill the bss segment          \n "
    " FillZerobss:                         \n"
    "     movs    r3, #0                   \n"
    "     str     r3, [r2], #4             \n"
    "                                      \n"
    " LoopFillZerobss:                     \n"
    "     ldr    r3, = _ebss               \n"
    "     cmp    r2, r3                    \n"
    "     bcc    FillZerobss               \n"
    "                                      \n"
    " @ Initialize system clock            \n"
    "     bl  SystemInit                   \n"
    "                                      \n"
    " @ Call static constructors           \n"
    "     bl __libc_init_array             \n"
    "                                      \n"
    " @ Enter main                         \n"
    "     bl    main                       \n"
    );
}
```

Esimerkkikoodi 9. Reset-poikkeuskäsittelijä.

Pääohjelman alussa, eli main-funktiossa, kutsutaan heti ensimmäisenä sysPlatformInit-funktiota (esimerkkikoodi 12), joka alustaa kaikki oheisjärjestelmäkomponentit. Funktiolle annetaan parametrina taulukon osoitin, joka sisältää kaikki ohjelmassa käytettävien oheisjärjestelmäkomponenttien alustusfunktiot (esimerkkikoodi 10).

```
typedef SYS_ERROR (*SYS_PLATFORM_INIT_FUNCTION)(SYS_PLATFORM_CONTEXT* ctx);

// Platform init type.

typedef struct
{
    SYS_PLATFORM_INIT_FUNCTION callback;
    const char*                decription;
}
SYS_INIT_PROCEDURE;

// Hardware initialization callbacks.

static const SYS_INIT_PROCEDURE sys_hw_init[]=
{
#if defined(CONFIG_TARGET_CUSTOM_V02)

    SYS_INIT_CALLBACK(stm32l4xx_system_clock_init,    "STM32L4 System Clock Init"),
    SYS_INIT_CALLBACK(stm32l4xx_ssd1306_display_init, "STM32L4 SSD1306 Display Init"),
    SYS_INIT_CALLBACK(stm32l4xx_tty_init,             "STM32L4 Console TTY Init"),
    SYS_INIT_CALLBACK(stm32l4xx_user_led_init,        "STM32L4 User Led Init"),
    SYS_INIT_CALLBACK(stm32l4xx_board_rtc_init,       "STM32L4 RTC Init"),
    SYS_INIT_CALLBACK(stm32l4xx_board_sdmmc_init,     "STM32L4 SDMMC Init"),
    SYS_INIT_CALLBACK(stm32l4xx_board_sensors_init,   "STM32L4 Sensor Init"),

#elif defined(CONFIG_TARGET_SIMULATOR)

    SYS_INIT_CALLBACK(simulator_tty_init,             "Posix Console TTY Init"),

#endif

    SYS_INIT_END
};
```

Esimerkkikoodi 10. Taulukko, joka pitää sisällään oheisjärjestelmäkomponenttien alustusfunktiot. Kyseisen taulukon osoitin annetaan parametrina sysPlatformInit-funktiolle, joka vastaa alustusfunktioiden kutsumisesta.

Oheisjärjestelmäkomponenttien käyttöön hyödynnetään STMicroelectronicsin STM32L4-abstraktiokerrosta. Esimerkkikoodi 12 havainnollistaa FTDI-mikropiirin kommunikointiin käytettävän UART-ohjaimen alustuksen. Alustuksen yhteydessä luodaan myös semafori, jonka avulla konsolisäie synkronoidaan. Samassa yhteydessä asetetaan oheisjärjestelmälaitekontekstin TTY-ajurin luku-, kirjoitus- ja odotusfunktiot, joita käytetään terminaali-ikkunan ja mikrokontrollerin väliseen kommunikointiin.

```
// tty_get_char

static SYS_ERROR tty_get_char(char* c)
{
    if ( rx_buffer_read_pos == DMA_BUFFER_INDEX(&hdma_rx) )
        return(EIO);

    *c= (char)rx_buffer[rx_buffer_read_pos];
    rx_buffer_read_pos++;

    if ( rx_buffer_read_pos >= UART_DMA_RX_BUFFER_SIZE )
        rx_buffer_read_pos= 0;
```

```
    return(0);
}

// tty_wait_char

static SYS_ERROR tty_wait_char(void)
{
    return(sysLockSemaphore(&tty_semaphore));
}

// stm32l4xx_tty_init

SYS_ERROR stm32l4xx_tty_init(SYS_PLATFORM_CONTEXT* board)
{
    SYS_ERROR rc;

    if ( !board )
        return(EINVAL);

    // UART configuration.

    uart_handle.Instance=                    UART_CONSOLE_INSTANCE;
    uart_handle.Init.BaudRate=               UART_CONSOLE_BAUDRATE;
    uart_handle.Init.WordLength=             UART_WORDLENGTH_8B;
    uart_handle.Init.StopBits=               UART_STOPBITS_1;
    uart_handle.Init.Parity=                 UART_PARITY_NONE;
    uart_handle.Init.Mode=                   UART_MODE_TX_RX;
    uart_handle.Init.HwFlowCtl=              UART_HWCONTROL_NONE;
    uart_handle.Init.OverSampling=           UART_OVERSAMPLING_16;
    uart_handle.Init.OneBitSampling=         UART_ONE_BIT_SAMPLE_DISABLE;
    uart_handle.AdvancedInit.AdvFeatureInit= UART_ADVFEATURE_NO_INIT;

    if ( HAL_UART_Init(&uart_handle) != HAL_OK )
        return(ENXIO);

    // Configure the DMA handler for reception process.

    hdma_rx.Instance=                 UART_CONSOLE_RX_DMA_CHANNEL;
    hdma_rx.Init.Direction=           DMA_PERIPH_TO_MEMORY;
    hdma_rx.Init.PeriphInc=           DMA_PINC_DISABLE;
    hdma_rx.Init.MemInc=              DMA_MINC_ENABLE;
    hdma_rx.Init.PeriphDataAlignment= DMA_PDATAALIGN_BYTE;
    hdma_rx.Init.MemDataAlignment=    DMA_MDATAALIGN_BYTE;
    hdma_rx.Init.Mode=                DMA_CIRCULAR;
    hdma_rx.Init.Priority=            DMA_PRIORITY_HIGH;
    hdma_rx.Init.Request=             UART_CONSOLE_RX_DMA_REQUEST;

    HAL_DMA_Init(&hdma_rx);

    // Associate the initialized DMA handle to the the UART handle.

    __HAL_LINKDMA(&uart_handle, hdmarx, hdma_rx);

    // NVIC configuration for DMA.

    HAL_NVIC_SetPriority(UART_CONSOLE_RX_DMA_IRQn, 3, 0);
    HAL_NVIC_EnableIRQ(UART_CONSOLE_RX_DMA_IRQn);

    // Read in DMA mode.

    if ( HAL_UART_Receive_DMA(&uart_handle, rx_buffer, UART_DMA_RX_BUFFER_SIZE) !=
HAL_OK )
        return(ENXIO);

    // Enable RX not empty -interrupt.

    __HAL_UART_ENABLE_IT(&uart_handle, UART_IT_RXNE);

    board->tty_driver.read= tty_get_char;
    board->tty_driver.put=  tty_put_char;
    board->tty_driver.wait= tty_wait_char;
```

```
    // Create tty semaphore.

    rc= sysCreateSemaphore(&tty_semaphore);
    if ( rc )
        return(rc);

    return(0);
}
```

Esimerkkikoodi 11. Konsolin UART-ohjaimen-alustus, sekä konsolisyötteen synkronointiin käytettävän semaforin alustus.

System-abstraktiokerros määrittelee tietorakenteen (esimerkkikoodi 12), jonka avulla hallitaan oheisjärjestelmälaitekutsuja. Kyseistä tietorakennetta voidaan ajatella myös oheisjärjestelmälaitteiden kontekstina ja siitä luodaan yksi instanssi ohjelman alussa, jonka osoitin annetaan jokaiselle ohjelman säikeelle.

```
// Sensor data type.

typedef struct
{
    int32_t sensor_info_bits;

    union
    {
        SYS_SENSOR_DATA_SHT31   sht31;
        SYS_SENSOR_DATA_LSM303C lsm303c;
        SYS_SENSOR_DATA_VL53L0X vl53l0x;
    }
    data;

}
SYS_SENSOR_DATA;

// Functions

typedef SYS_ERROR (*HV_INPUT_READ_FUNCTION)(SYS_HV_INPUT_STATUS* status,
                                            SYS_HV_CHANNEL       input_channel);

typedef SYS_ERROR (*LED_FUNCTION)(LED_COMMAND cmd);

typedef SYS_ERROR (*SD_STATE_READ_FUNCTION)(SD_CARD_STATUS* status);

typedef SYS_ERROR (*SYS_PLATFORM_INIT_FUNCTION)(SYS_PLATFORM_CONTEXT* ctx);

typedef SYS_ERROR (*SYS_GET_DATE_TIME_FUNCTION)(SYS_TIMEDATE* time_date);

typedef SYS_ERROR (*SYS_SET_DATE_TIME_FUNCTION)(SYS_TIMEDATE time_date);

typedef SYS_ERROR (*SYS_SENSOR_READ_FUNCTION)(SYS_SENSOR         sensor_type,
                                              SYS_SENSOR_ADDRESS sensor_address,
                                              SYS_SENSOR_DATA*   sensor_data);

// Wi-Fi context type.

typedef struct
{
    int32_t status;
    int32_t socket;
    char    ap_pass[WIFI_MAX_PASS_LENGHT];
    char    ap_ssid[WIFI_MAX_SSID_LENGHT];
    char    ip_addr[WIFI_MAX_IP_ADDRESS_LENGTH];
    char    mac_addr[WIFI_MAX_MAC_ADDRESS_LENGTH];
```

```
}
SYS_WIFI_CONTEXT;

// Platfrom context type.

typedef struct system_platform_context
{
    SYS_GET_DATE_TIME_FUNCTION get_time_func;
    SYS_SET_DATE_TIME_FUNCTION set_time_func;
    SD_STATE_READ_FUNCTION     sd_card_status_func;
    SYS_SENSOR_READ_FUNCTION   sensor_read_func;
    LED_FUNCTION               user_led_func;
    SYS_TTY_DRIVER             tty_driver;
    const SYS_WIFI_CONTEXT*    wifi_ctx;
    const void*                sd_driver;
    void*                      display_driver;
}
SYS_PLATFORM_CONTEXT;
```

Esimerkkikoodi 12. Oheisjärjestelmäkomponenttien kontekstiin liittyvät tietorakenne- ja funktiotyypit.

Oheisjärjestelmäkomponenttien alustuksen jälkeen luodaan ohjelman säikeet ja käynnistetään ytimen vuorottaja (esimerkkikoodi 13).

```
// main

int main(void)
{
    SYS_ERROR rc;

    // Initialize platform.

    rc= sysPlatformInit(&sys_hw_context, sys_hw_init);
    SYS_ASSERT(!rc);

#if defined(CONFIG_TARGET_CUSTOM_V02)

    // Create sensor thread.

    rc= sysCreateThread(sensor_thread, 0, "SENSOR", SENSOR_THREAD_SIZE, PRIORITY_MEDIUM,
&sys_hw_context);
    SYS_ASSERT(!rc);

#endif

    // Create console thread.

    rc= sysCreateThread(console_thread, 0, "CONSOLE", SHELL_THREAD_SIZE, PRIORITY_HIGH,
&sys_hw_context);
    SYS_ASSERT(!rc);

#if defined(CONFIG_USE_WIFI_UART) || defined(CONFIG_USE_WIFI_SPI)

    // Create WiFi thread.

    rc= sysCreateThread(wifi_thread, 0, "WIFI", WIFI_THREAD_SIZE, PRIORITY_HIGHEST,
&sys_hw_context);
    SYS_ASSERT(!rc);

#endif

#if defined(CONFIG_USE_SDMMC)

    // Create SD thread.
```

```
    rc= sysCreateThread(sdmmc_thread, 0, "SD", SD_THREAD_SIZE, PRIORITY_MEDIUM,
&sys_hw_context);
    SYS_ASSERT(!rc);

#endif

#if defined(CONFIG_USE_OLED)

    // Create display thread.

    rc= sysCreateThread(display_thread, 0, "DISPLAY", DISPLAY_THREAD_SIZE, PRIORITY_ME-
DIUM, &sys_hw_context);
    SYS_ASSERT(!rc);

#endif

    // Start scheduler.

    sysStartScheduler();

    return(0);
}
```

Esimerkkikoodi 13. Ohjelman main-funktio, jossa alustetaan säikeet.

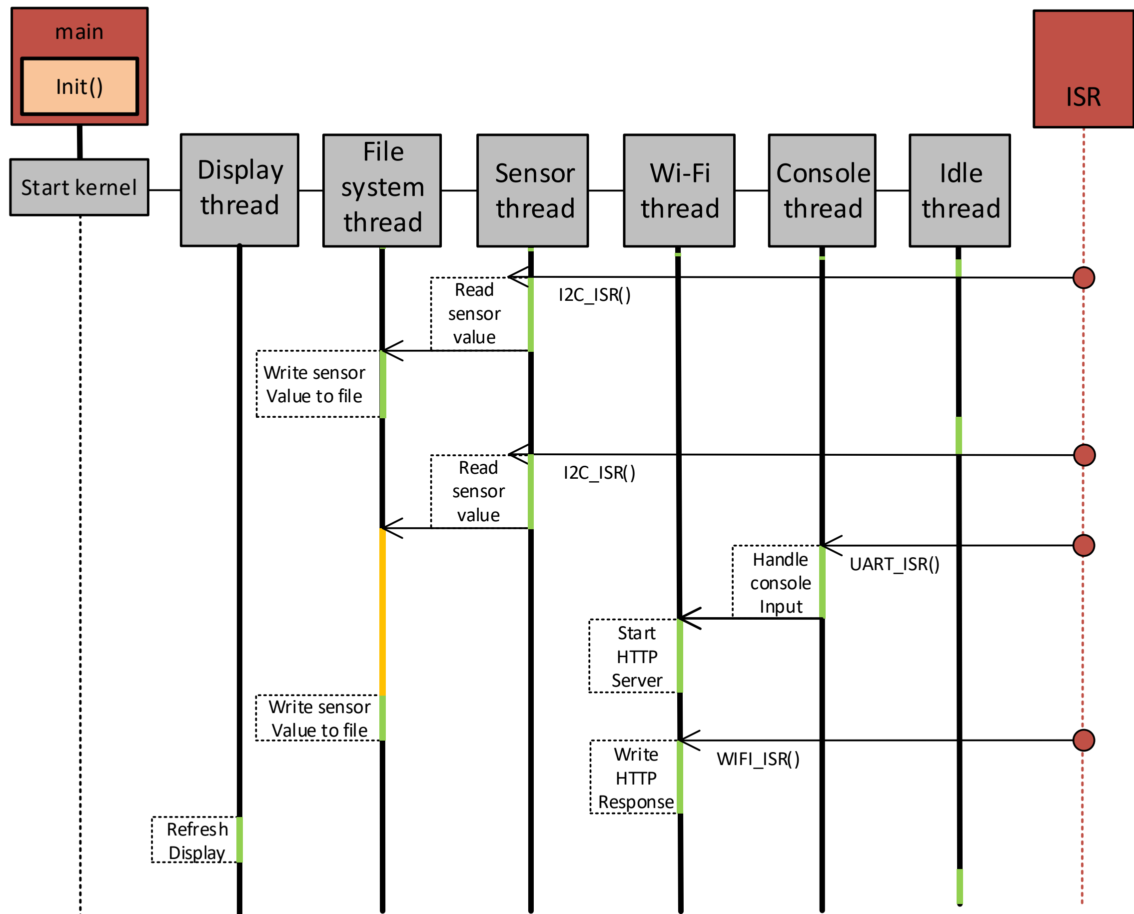
5.4 Säikeet

Laitteen ohjelmisto koostuu yhteensä viidestä eri säikeestä, jotka alustetaan ja käynnistetään pääohjelman alussa. Jokaisella säikeellä on oma vastuualue, jotka ilmenevät taulukosta 4.

Taulukko 4. Ohjelman säikeet ja niiden tehtävät.

Säie	**Tehtävä**	**Suoritusehto**
sensor_thread	Laitteeseen kytkettyjen sensoreiden arvon lukeminen joko GPIO-keskeytyksen kautta, tai määräajan kuluessa.	I^2C-ohjaimelta tuleva keskeytys tai määräajan saavuttaminen.
wifi_thread	Suorittaa kommunikoinnin Wi-Fi-moduulin ja mikrokontrollerin välillä, sekä vastaa myös HTTP-serverin toiminnasta.	UART- tai SPI-ohjaimelta tuleva keskeytys tai console_thread-säikeeltä tuleva signaali.
sdmmc_thread	Suorittaa tiedostojärjestelmän luku- ja kirjoitusoperaatioita.	Signaali, joko sensor_thread- tai console_thread-säikeeltä.
console_thread	Komentokonsolin suoritus.	UART- tai DMA-ohjaimelta tuleva keskeytys,
display_thread	Laitteeseen kytketyn näytön piirto-operaatiot (laitteen tilan	Määräaika.

Säikeet jakavat suoritinaikaa tapahtumapohjaisesti kuvan 17 kaavion mukaisesti. Säikeen suoritukseen vapauttava tapahtuma voi olla oheisjärjestelmäkomponentilta tuleva keskeytys, toiselta säikeeltä vastaanotettu signaali tai määräajan saavuttaminen.



Kuva 17. Säikeiden välinen suoritinajan jakaminen.

5.5 Konsolikäyttöliittymä

Laitteeseen kehitettiin konsolikäyttöliittymä, jota voidaan käyttää esimerkiksi Unix-pohjaisessa käyttöjärjestelmässä screen-komentorivityökalun avulla (kuva 18).

```
File  Edit  View  Search  Terminal  Help

##########################################################
******************* CONSOLE COMMANDS *******************
##########################################################


fs-rw-test                    : SD card R/W test.
fs-list-files                 : List all files.
fs-print-file                 : Print file content. <file path>
fs-format                     : Format SD-card.
led-off                       : Set user LED off.
led-on                        : Set user LED on.
sensor-value                  : Display sensor value. <sensor>
channel-value                 : Display 24 V channel value. <channel>
display-test                  : Run display test.
display-off                   : Set display off.
display-on                    : Set display on.
display-invert                : Invert display colors.
wifi-ap-connect               : Connect to AP. <SSID> <PASSWORD>
wifi-ap-create                : Create AP.     <SSID> <PASSWORD>
wifi-dumb-buffer              : Dump I/O buffer.
wifi-print-address            : Print MAC/IP address.
wifi-start-server             : Start server.
wifi-stop-server              : Stop server.
wifi-list-ap                  : List access points.
kernel-stats                  : Print run-time stats.
rtc-get-time                  : Print RTC time.
rtc-set-date                  : Set RTC date. <DD> <MM> <YYYY>
rtc-set-time                  : Set RTC time. <HH> <MM> <SS>
sys>
```

Kuva 18. Kuvakaappaus terminaali-ikkunasta, jossa suoritetaan screen-ohjelmaa laitteen konsolikäyttöliittymän ja tietokoneen väliseen kommunikointiin.

Konsolikäyttöliittymä toimii siten, että ohjelmakoodiin määritellään komento ja komentoa vastaava takaisinkutsufunktio (esimerkkikoodi 14). Konsolisäikeen päällä ajetaan komentorivitulkkia, joka lukee asynkronisesti FTDI-piiriltä tulevaa dataa, joka on tässä tapauksessa komentorivin näppäinpainallukset. Mikäli syötetty komento tunnistetaan, suoritetaan komentoa vastaava takaisinkutsufunktio.

```
// Shell macros.

#define SYS_SHELL_ARGC argc
#define SYS_SHELL_ARGV argv

#define SYS_SHELL_ARGS_UNUSED() \
    (void)(SYS_SHELL_ARGC);     \
    (void)(SYS_SHELL_ARGV);

#define SYS_SHELL_CALLABCK(FUNCTION) \
    SYS_ERROR FUNCTION(const char** SYS_SHELL_ARGV, int32_t SYS_SHELL_ARGC)

// Shell command list

static const SYS_SHELL_COMMAND commands[]=
{
#if defined(CONFIG_USE_SDMMC)
    SYS_SHELL_COMMAND(CMD_SD_CARD_RW_TEST,       "fs-rw-test",          "SD card R/W test.", 0, 0),
    SYS_SHELL_COMMAND(CMD_SD_CARD_LIST_FILES,    "fs-list-files",       "List all files.", 0, 0),
    SYS_SHELL_COMMAND(CMD_SD_CARD_PRINT_FILE,    "fs-print-file",       "Print file content. <file path>", 1,
1),
    SYS_SHELL_COMMAND(CMD_SD_CARD_FORMAT,        "fs-format",           "Format SD-card.", 0, 0),
#endif

#if defined(CONFIG_TARGET_SIMULATOR)
    SYS_SHELL_COMMAND(CMD_EXIT,                  "exit",                "Exit.", 0, 0),
#endif

#if defined(CONFIG_TARGET_CUSTOM_V02)
    SYS_SHELL_COMMAND(CMD_USER_LED_OFF,          "led-off",             "Set user LED off.", 0, 0),
    SYS_SHELL_COMMAND(CMD_USER_LED_ON,           "led-on",              "Set user LED on.", 0, 0),
    SYS_SHELL_COMMAND(CMD_DISPLAY_SENSOR_VALUE,  "sensor-value",        "Display sensor value. <sensor>", 1,
1),
    SYS_SHELL_COMMAND(CMD_DISPLAY_INPUT_CHANNEL, "channel-value",       "Display 24 V channel value. <chan-
nel>", 1, 1),
#endif

#if defined(CONFIG_USE_OLED)
    SYS_SHELL_COMMAND(CMD_DISPLAY_TEST,          "display-test",        "Run display test.", 0, 0),
    SYS_SHELL_COMMAND(CMD_DISPLAY_OFF,           "display-off",         "Set display off.", 0, 0),
    SYS_SHELL_COMMAND(CMD_DISPLAY_ON,            "display-on",          "Set display on.", 0, 0),
    SYS_SHELL_COMMAND(CMD_DISPLAY_INVERT,        "display-invert",      "Invert display colors.", 0, 0),
#endif

#if defined(CONFIG_USE_WIFI_UART) || defined(CONFIG_USE_WIFI_SPI)
    SYS_SHELL_COMMAND(CMD_WIFI_CONNECT,          "wifi-ap-connect",     "Connect to AP. <SSID> <PASSWORD>",
2, 1),
    SYS_SHELL_COMMAND(CMD_WIFI_CREATE_AP,        "wifi-ap-create",      "Create AP.     <SSID> <PASSWORD>",
2, 1),
    SYS_SHELL_COMMAND(CMD_WIFI_DUMP_BUFFER,      "wifi-dump-buffer",    "Dump I/O buffer.", 0, 0),
    SYS_SHELL_COMMAND(CMD_WIFI_GET_ADDRESS,      "wifi-print-address",  "Print MAC/IP address.", 0, 0),
    SYS_SHELL_COMMAND(CMD_WIFI_START_SERVER,     "wifi-start-server",   "Start server.", 0, 0),
    SYS_SHELL_COMMAND(CMD_WIFI_STOP_SERVER,      "wifi-stop-server",    "Stop server.", 0, 0),
    SYS_SHELL_COMMAND(CMD_WIFI_LIST_APS,         "wifi-list-ap",        "List access points.", 0, 0),
#endif

#if defined(CONFIG_USE_FREERTOS)
    SYS_SHELL_COMMAND(CMD_KERNEL_STATS,          "kernel-stats",        "Print run-time stats.", 0, 0),
#endif

#if defined(CONFIG_USE_RTC)
    SYS_SHELL_COMMAND(CMD_RTC_GET_TIMESTAMP,     "rtc-get-time",        "Print RTC time.", 0, 0),
    SYS_SHELL_COMMAND(CMD_RTC_SET_DATE,          "rtc-set-date",        "Set RTC date. <DD> <MM> <YYYY>", 3,
3),
    SYS_SHELL_COMMAND(CMD_RTC_SET_TIME,          "rtc-set-time",        "Set RTC time. <HH> <MM> <SS>", 3,
3),
#endif
};
```

Esimerkkikoodi 14. Ohjelman komentorivikomentojen määrittely.

Komentokonsoli on suunniteltu sillä periaatteella, että takaisinkutsufunktioilla ei tehdä mitään muuta kuin signaloidaan säiettä, joka vastaa komennon suorituksesta. Tarkoituksena on, että konsolisäie on aina vapaana ohjelmansuoritukseen ja lukemaan uutta komentoa. Esimerkiksi saatavilla olevien Wi-Fi-verkkojen listaukseen käytetty funktio estää

säikeen ohjelmansuorituksen muutamien sekuntien ajaksi, jolloin seuraavan konsolikomennon käsittely saattaisi viivästyä (esimerkkikoodi 15).

```
// wifi_thread

void wifi_thread(void* arg)
{
    SYS_THREAD_SIGNAL event;
    SYS_ERROR         rc;

    // ...

    while ( 1 )
    {

        // Wait for events.

        rc= sysSignalWait(WIFI_TASK_CONNECT_TO_AP |
                          WIFI_TASK_CREATE_AP     |
                          WIFI_TASK_PRINT_ADDRESS |
                          WIFI_TASK_START_SERVER  |
                          WIFI_TASK_STOP_SERVER   |
                          WIFI_TASK_LIST_AP       |
                          WIFI_TASK_RX_ISR_SIGNAL, &event, 0);
        if ( rc )
        {
            sysLogError(rc, SYS_ERROR_MSG_THREAD_API, 0);
            sysKillCurrentThread();
        }

        // List access points.

        if ( (event & WIFI_TASK_LIST_AP) )
        {
            wifi_list_aps(&wifi_ctx);
        }

        // ...
      }
}

// CMD_WIFI_LIST_APS

SYS_SHELL_CALLABCK(CMD_WIFI_LIST_APS)
{
    SYS_ERROR rc;

    SYS_SHELL_ARGS_UNUSED();

    // Notify Wi-Fi thread to list all available
    // access points.

    rc= sysSignalThread(&wifi_thread_handle, WIFI_TASK_LIST_AP);
    if ( rc )
    {
        sysLogError(rc, SYS_ERROR_MSG_THREAD_API, 0);
        return(rc);
    }

    return(0);
}
```

Esimerkkikoodi 15. Otanta Wi-Fi-verkkojen listaukseen käytetyn komennon käsittelystä.

6 Yhteenveto

Tiedonkeräimen suunnittelussa ja toteutuksessa onnistuttiin ja toiminnalliset tavoitteet saavutettiin suurimmalta osin. Laitteen käyttövarmuus osoittautui ongelmalliseksi, sillä laite ei aina vastannut palvelupyyntöihin ja tiedostojen luku- ja kirjoitusoperaatioissa ilmeni myös ajoittain virheitä.

Vaikka kehitetty laite ei ole toiminnallisuudeltaan täydellinen, se ei ollut myöskään tämän projektin päällimmäinen tavoite. Tavoite oli päästä käytännössä toteuttamaan jonkinasteinen mikrokontrolleripohjainen laite ja laitteen varsinainen toiminnallisuus valikoitui lähes sattumalta. Tässä projektissa kehitetylle laitteelle ei ole minkäänlaista toiminnallista tarvetta eikä laitteen jatkokehitykseen laiteta enää aikaa. Toisin sanoen, projektissa kehitetty laite toimii tämän projektin päätteeksi verkonpainona.

Lähteet

1 ARMv7-M Architecture Reference Manual (ARM DDI 0403E.b). 2014. Datalehti. ARM. <https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf>

2 ARM Cortex-M4 Processor Technical Reference Manual Revision r0p1. 2015. Datalehti. ARM. <https://static.docs.arm.com/100166/0001/arm_cortexm4_processor_trm_100166_0001_00_en.pdf>

3 Synchronous and asynchronous exceptions. Verkkoaineisto. ARM Developer. <https://developer.arm.com/docs/100933/latest/synchronous-and-asynchronous-exceptions>. Luettu 25.10.2019.

4 Langbridge, James A. 2014. Professional Embedded ARM Development. USA: John Wiley & Sons.

5 Procedure Call Standard for the ARM Architecture (ARM IHI 0042F). 2015. Datalehti. ARM. <http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf>

6 The Institute of Electrical and Electronics Engineers, Inc. & The Open Group. Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7 (C082). 2008. UK: The Open Group.

7 Barry, Richard. Mastering the FreeRTOS Real Time Kernel, Pre-release 161204 Edition. E-kirja. Real Time Engineers Ltd.

8 Cortex-M4 Devices Generic User Guide (ARM DUI0553B). 2011. Datalehti. ARM.<http://infocenter.arm.com/help/topic/com.arm.doc.dui0553b/DUI0553.pdf>

9 The BBC Microcomputer and me, 30 years down the line. Verkkoaineisto. The British Broadcasting Company. <https://www.bbc.com/news/technology-15969065>. Päivitetty 1.12.2011. Luettu 25.10.2019.

10 Lamie, Edward L. 2005. Real-Time Embedded Multithreading: Using ThreadX and ARM. USA: CMP Books.

11 Yiu, Joseph. 2009. The Definitive Guide to the ARM Cortex-M3 (2nd Edition). USA: Newnes Books.

12 Yiu, Joseph. 2015. The Definitive Guide to the ARM Cortex-M0 and Cortex-M0+ Processors (2nd Edition). USA: Newnes Books.

13 Furber, Steve. 2000. ARM System-on-Chip Architecture (2nd Edition). USA: Addison-Wesley Professional.

14 Sloss, Andrew N. & Symes, Dominic & Wright, Chris. ARM System Developer's Guide – Designing and Optimizing System Software (1st Edition). 2014. NL: Elsevier Inc.

15 Universal Serial Bus Specification Revision 2.0. 2000. Datalehti. USB Implementers Forum, Inc. <https://www.usb.org/document-library/usb-20-specification>

16 Ultra-low-power Arm Cortex-M4 32-bit MCU+FPU (DS12024, Revision 3). 2018. Datalehti. STMicroelectronics. <https://www.st.com/resource/en/datasheet/stm32l4s5vi.pdf>

17 Getting started with STM32L4 Series and STM32L4+ Series hardware development (AN4555, Revision 7). 2018. Datalehti. STMicroelectronics. <https://www.st.com/resource/en/application_note/dm00125306.pdf>

18 Low-power timer (LPTIM) applicative use-cases on STM32 microcontrollers (AN4865, Revision 4). 2019. Datalehti. STMicroelectronics. <https://www.st.com/resource/en/application_note/dm00290631.pdf>

19 Bai, Ying. 2016. Practical Microcontroller Engineering with ARM Technology. USA: John Wiley & Sons.

20 Nieminen, Paavo. 2019. ITKA203-kurssimateriaali. Verkkodokumentti. Jyväskylän yliopisto. <http://users.jyu.fi/~nieminen/kj19/kayttojarjestelmat.pdf>. Päivitetty 20.3.2019. Luettu 25.10.2019.

21 Cortex-M4(F) Lazy Stacking and Context Switching (ARM DAI0298A). 2012. Datalehti. ARM. <https://static.docs.arm.com/dai0298/a/DAI0298A_cortex_m4f_lazy_stacking_and_context_switching.pdf>

22 Remes, Alexandre. 2013. Arm Fundamentals: Introduction to understanding Arm processors. Verkkoaineisto. ARM Community. <https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-fundamentals-introduction-to-understanding-arm-processors>. Päivitetty 11.9.2013. Luettu 25.10.2019.

23 Coombs, Clyde F. 2007. Printed Circuits Handbook (6th Edition). USA: McGraw-Hill Education.

24 Zurell, Kirk. 2000. C Programming for Embedded Systems. USA: CMP Books.

25 Liu, Jane W. S. Real-Times System. 2000. USA: Prentice-Hall.

26 Qing, Li & Yao, Caroline. Real-Time Concepts for Embedded Systems. 2003. USA: CMP Books.

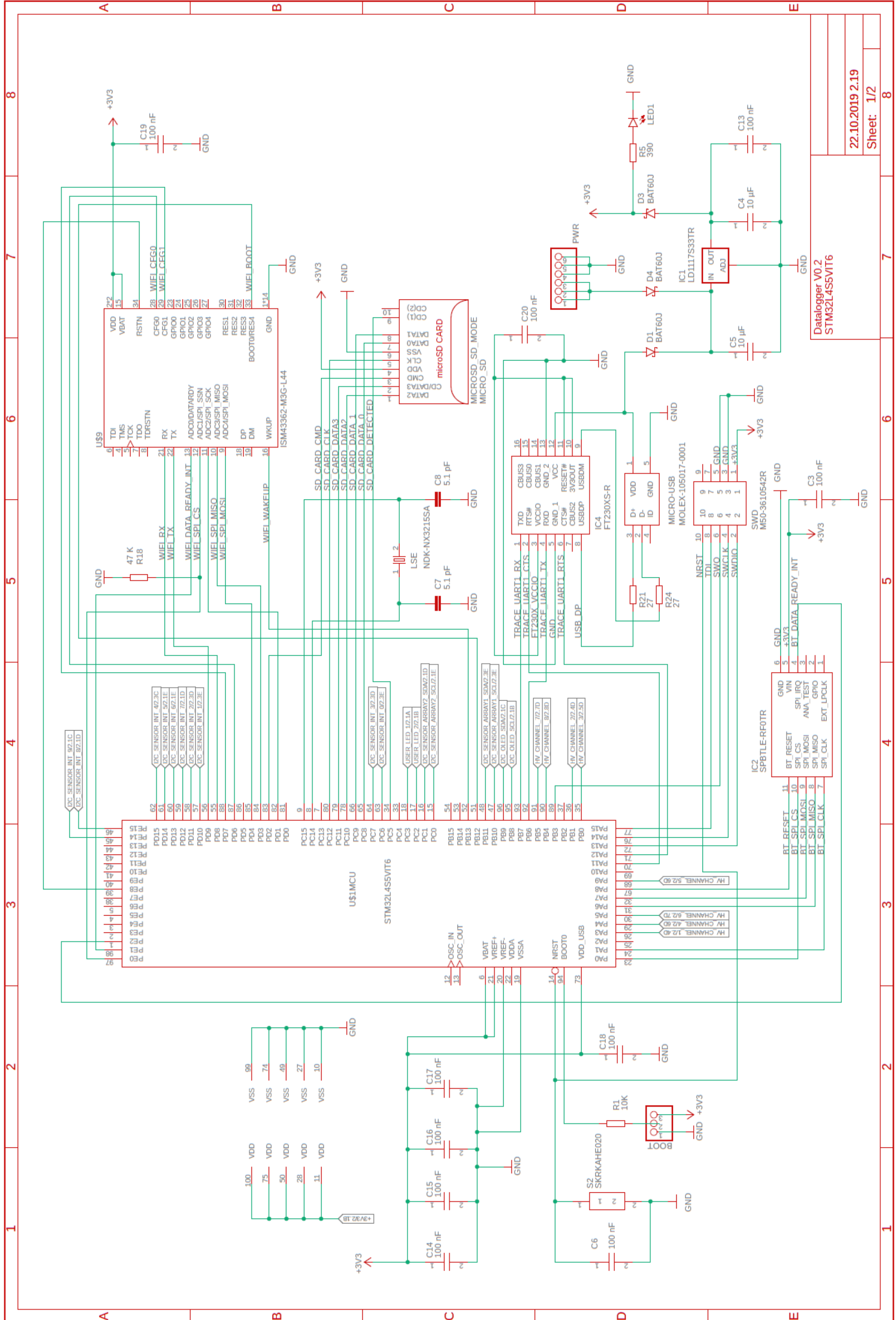
27 Puhan, Janez. Operating System, Embedded System, and Real-Time Systems. 2015. SL: FE Publishing.

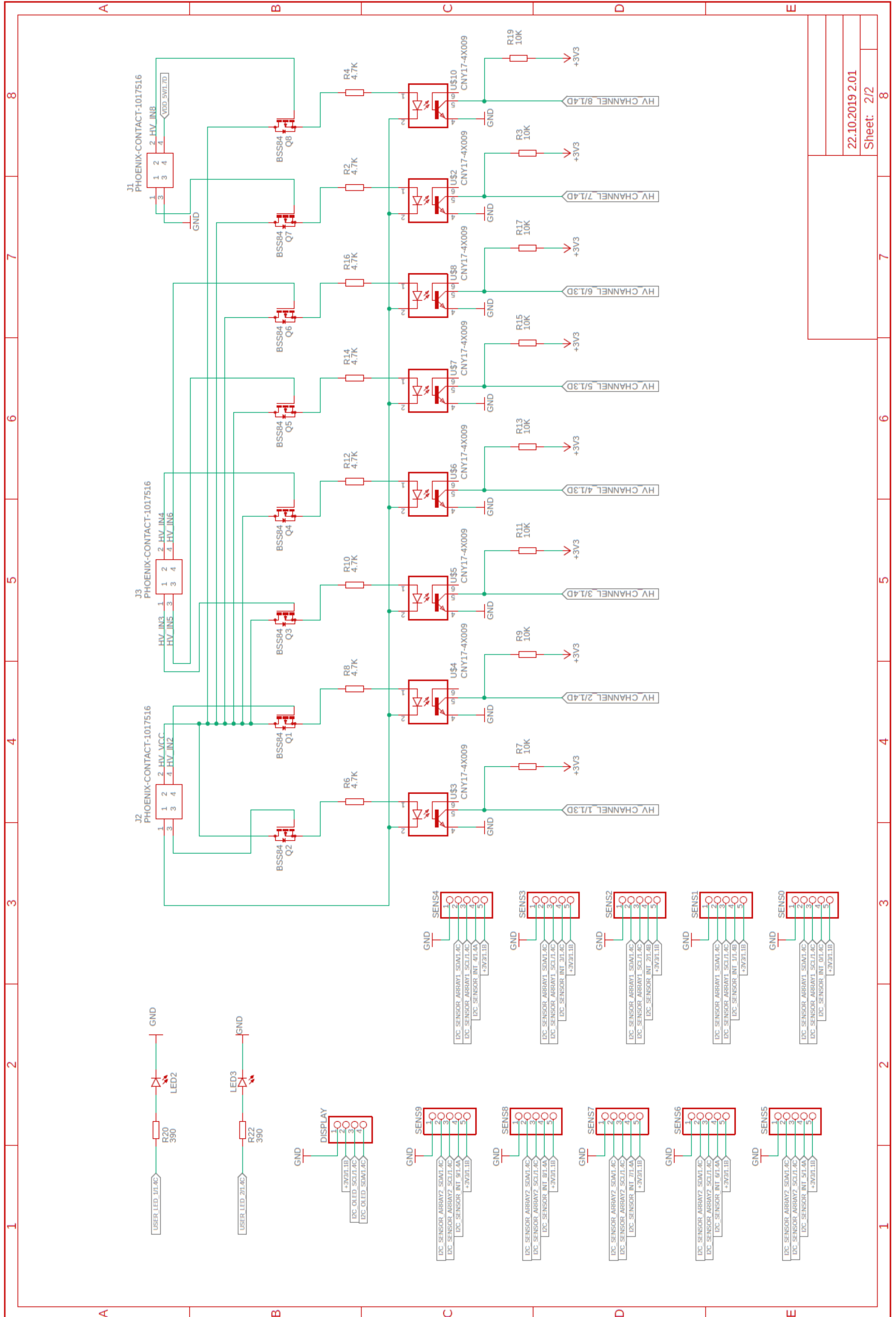
28 Fan, Xiaocong. Real-Time Embedded Systems – Design Principles and Engineering Practices. 2015. NL: Elsevier Inc.

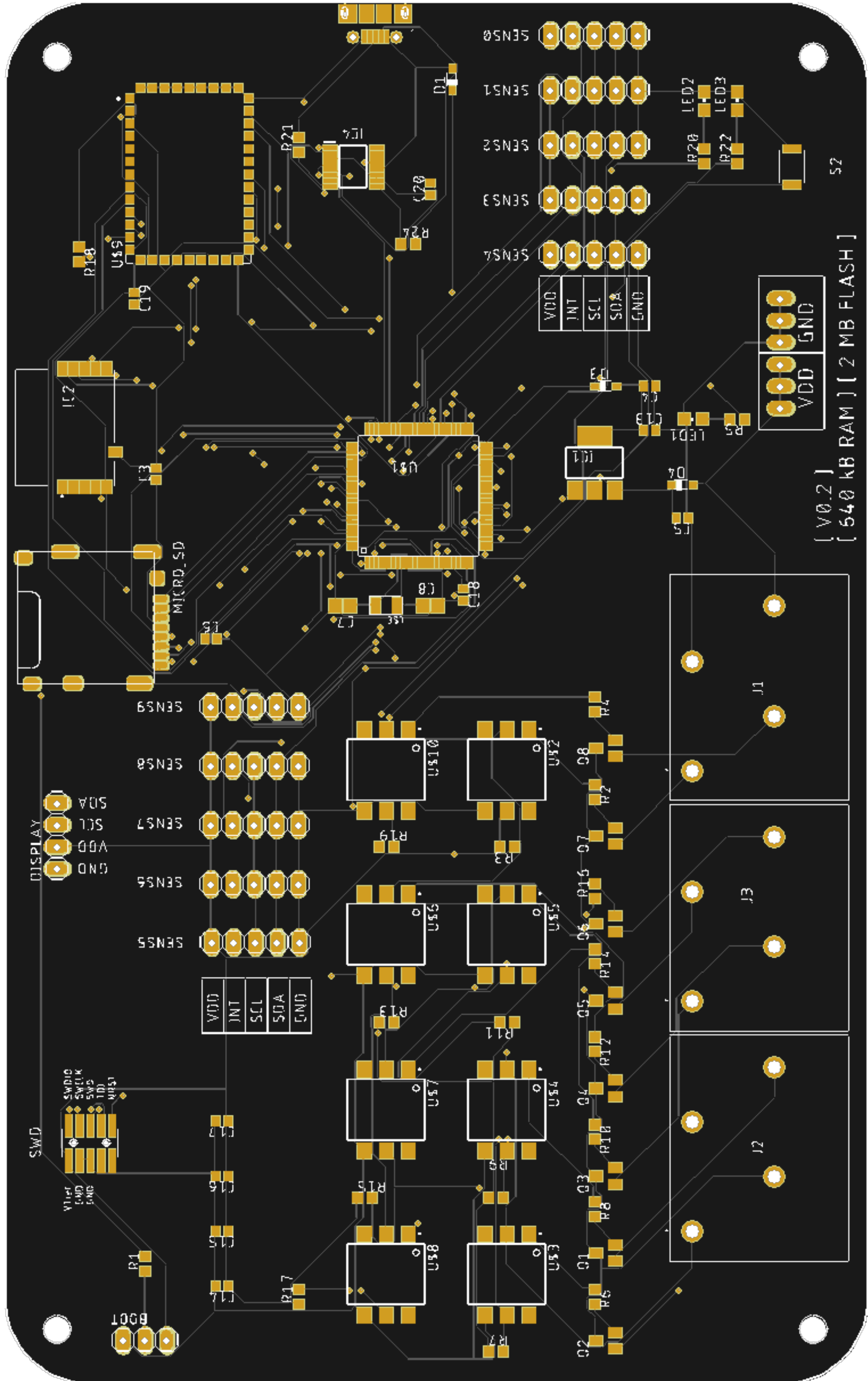
29 Gatliff, Bill. Porting and Using Newlib in Embedded Systems. 2001. Verkkoaineisto. <http://www.billgatliff.com/newlib.html>

Part	Value	Device	Package	Manufacturer	Part Number
C3, C6, C13, C14, C1	100 nF	Multilayer Ceramic Capacitor	0603 (1608 metric)	Kemet	C0603C104K4RECTU
C4, C5	10 µF	Multilayer Ceramic Capacitor	0603 (1608 metric)	Kemet	C0603C106M8PACTU
C7, C8	5.1 pF	Multilayer Ceramic Capacitor	0805 (2012 metric)	Kemet	C0805C519C5GACTU
R1, R3, R7, R9, R11,	10K	Resistor	0201 (0603 metric)	Vishay	CRCW020110K0FNED
R2, R4, R6, R8, R10,	4.7K	Resistor	0201 (0603 metric)	Vishay	CRCW02014K70FKED
R5, R20, R22	390	Resistor	0201 (0603 metric)	Vishay	CRCW0201390RJNED
R18	47K	Resistor	0201 (0603 metric)	Vishay	CRCW020147K0FKED
R21, R24	27	Resistor	0201 (0603 metric)	Vishay	CRCW020127R0FNED
LED1, LED2, LED3		LED	0805 (2012 metric)	Dialight	599-0130-007F
D1, D3, D4		Schottky Diode/Rectifier	SOD-323	STMicroelectronics	BAT60JFILM
IC1		LDO Voltage Regulator	SOT-223-3	STMicroelectronics	LD1117S33TR
IC2		Bluetooth Module	SPSGRF868	STMicroelectronics	SPBTLE-RF0TR
IC4		USB Interface IC	SSOP-16	FTDI Chip	FT230XS-R

Part	Device	Package	Manufacturer	Part Number
J1, J2, J3	Fixed Terminal Block		Phoenix Contact	1017516
U$1	Microcontroller	LQFP100	STMicroelectronics	STM32L4S5VIT6
U$9	WiFi Module	ISM43362-M3G-L44	Inventek Systems	ISM43362-M3G-L44-E-C3.5.2.5
U$2, U$3, U$4, U$5, U$6, U$7, U$8, U$10	Optocoupler	SOIC254P1016X390-6N	Vishay	CNY17-4X009
LSE	Crystal		NDK	NX3215SA-32.768K-STD-MUA-1
MICRO-USB	USB Connector		Molex	105017-0001
MICRO_SD	Memory Card Connector		Molex	502774-0891
Q1-Q8	MOSFET	SOT23	Nexperia	BSS84,215
SWD	M50-3610542R	M503610542	HARWIN	M50-3610542R
S2	Tactile Switch		ALPS	SKRKAHE020
BOOT		1X03		
DISPLAY		1X04		
PWR		1X06		
SENS0-SENS9		1X05		


Datalogger V0.2
STM32L4S5VIT6
22.10.2019 2.19
Sheet: 1/2


PHOENIX-CONTACT-1017516
CNY17-4X009
BSS84
22.10.2019 2.01
Sheet: 2/2


SENS0
SENS1
SENS2
SENS3
SENS4
SENS5
SENS6
SENS7
SENS8
SENS9
VDD
INT
SCL
SDA
GND
[V0.2]
[540 kB RAM] [2 MB FLASH]
MICRO_SD
DISPLAY
SWD
BOOT
J1
J2
J3
S2