Vladimir Byckling

Ilmalämpöpumpun ohjauskortin kehittäminen Linux-alustalle

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

2.9.2013

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Vladimir Byckling Ilmalämpöpumpun ohjauskortin kehittäminen Linux-alustalle 33 sivua + 19 liitettä 2.9.2013
Tutkinto	insinööri (AMK)
Koulutusohjelma	tietotekniikka
Suuntautumisvaihtoehto	sulautettu tietotekniikka
Ohjaaja(t)	yliopettaja Markku Nuutinen

Insinöörityön tavoite oli suunnitella ja toteuttaa prototyyppi ohjainkortista, jota voitaisiin käyttää Mitsubishi MSZ-GE25VA -merkkisen ilmalämpöpumpun säätöön ja etäohjaukseen. Työn kohteena olevan ilmalämpöpumpun ohjaus tapahtui infrapunavaloa lähettävällä kaukosäätimellä ja tämä ratkaisu haluttiin laajentaa osaksi sulautettua järjestelmää, jonka tarkoitus on tarjota etähallinta- sekä muita automatisoituja toimintoja.

Kehitystyön lähtökohdaksi valittiin sulautettu Linux-pohjainen tietokonekortti, johon ohjainkortti tultaisiin liittämään. Ohjainkortin toteutukseen asetettiin ehtoja, joissa pyrittiin säilyttämään ohjainkortin järjestelmäriippumattomuus ja esteettömyys. Toteutuksen ehdoilla pyrittiin takaamaan ohjainkortin uudelleenkäytettävyys muissa alustoissa ja vaivaton loppusijoittelu ohjaimena.

Ohjainkortin kehitys aloitettiin kartoittamalla järjestelmän kohdeympäristöä, johon valittiin tietokonekortiksi Raspberry PI Arch Linux -käyttöjärjestelmällä sekä asetettiin lähtökohdat ohjainkortin kehitykselle. Ohjainkortin kehityksen tueksi työn teoriaosuudessa käsiteltiin aiheet RS-232-standardista, SPI-protokollasta, optisesta tiedonsiirrosta sekä Linux-laiteajureista.

Lopputuloksena ohjainkortin kehitykselle aikaansaatiin ohjainkortin prototyyppi, jolla pystyttiin osoittamaan prototyypin perustoiminta. Täysin toimintavalmista laitetta ei toteutettu, mutta kehitystyön tulos asetti pohjan ohjainkortin jatkokehitystyölle.

Avainsanat	mikro-ohjain, sulautettu järjestelmä, Linux, Linux-laiteajurit, Raspberry Pi

Author(s) Title Number of Pages Date	Vladimir Byckling Controller board development of air-source heat pump for Linux platform 33 pages + 19 appendices 2 September 2013
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Embedded Systems
Instructor(s)	Markku Nuutinen, Principal Lecturer

The objective of the final year project was to design and implement a prototype of a controller board which could be used for adjusting and remote controlling a Mitsubishi MSZ-GE25VA branded air source heat pump. The pump used in the project was operated by an infrared light-emitting remote controller, and this solution was meant to be integrated into an embedded system which would provide remote controlling capabilities on its own with other automated solutions.

The starting point of the controller board development was an embedded computer with the Linux operating system as a target platform, where the controller board would be attached to operate as a part of the system. The controller board development prerequisites included a platform independent solution, where the controller board could be reused in other platforms as needed and at the same enable easy installation.

The development cycle started by identifying the target environment, which included selecting Raspberry Pi as the embedded computer solution with the Arch Linux operating system to further identify necessary parts of the controller board development. To support the development, the theoretical section of the thesis describes the RS-232 standard, an SPI protocol, optical data transmission methods, and finally Linux device drivers.

As a result of the controller board development, a working prototype was implemented which was capable of providing basic functionalities. A fully functional device was not implemented, but the prototype development provided ground work for further development.

Keywords	microcontroller, embedded system, Linux, Linux device drivers, Raspberry Pi

Metropolia

Sisällysluettelo

1 Johdanto 1

2 Kohdeympäristö, laitteet ja sovellukset 2

3 RS-232-sarjaportti 5

4 SPI-protokolla 7

5 Optinen tiedonsiirto 8

6 Linux-laiteajurit 10

6.1 Linux 10
6.2 Laiteajurien tehtävä 11
6.3 Laiteajurin runko 13
6.4 Laiteajurin kääntäminen 14
6.5 Laiteajurin liittäminen ytimeen 15
6.6 Laiteajurit ja käyttäjäavaruus 16
6.6.1 Laitetiedostot 16
6.6.2 Proc 17
6.6.3 Unified Device Model ja Sysfs 18

7 IR-ohjainkortin toteutus 23

7.1 Lohkokaavio 23
7.2 IR-ohjainkortin toimintakaavio 26
7.3 IR-ohjainkortin piirikaavio ja sähköiset ominaisuudet 27
7.4 Mikro-ohjainten ohjelmakoodi 27
7.5 Ohjainkortin testaus 29
7.6 IR-ohjainkortin jatkokehitys 30

8 Yhteenveto 30

Lähteet 32

Liitteet

Liite 1. IR-ohjainkortin lähettimen toimintakaavio

Liite 2. IR-ohjainkortin ohjaimen toimintakaavio

Liite 3. IR-ohjainkortin piirikaavio

Liite 4. IR-ohjainkortin CC2500.h-tiedoston lähdekoodi

Liite 5. IR-ohjainkortin CC2500.c-tiedoston lähdekoodi

Liite 6. IR-ohjainkortin RadioModule.h-tiedoston lähdekoodi

Liite 7. IR-ohjainkortin RadioModule.c-tiedoston lähdekoodi

Liite 8. IR-ohjainkortin ohjaimen Interrupts.h-tiedoston lähdekoodi

Liite 9. IR-ohjainkortin ohjaimen Interrupts.c-tiedoston lähdekoodi

Liite 10. IR-ohjainkortin ohjaimen IRModule.h-tiedoston lähdekoodi

Liite 11. IR-ohjainkortin ohjaimen IRModule.c-tiedoston lähdekoodi

Liite 12. IR-ohjainkortin ohjaimen main.h-tiedoston lähdekoodi

Liite 13. IR-ohjainkortin ohjaimen main.c-tiedoston lähdekoodi

Liite 14. IR-ohjainkortin lähettimen Console.h-tiedoston lähdekoodi

Liite 15. IR-ohjainkortin lähettimen Console.c-tiedoston lähdekoodi

Liite 16. IR-ohjainkortin lähettimen Interrupts.h-tiedoston lähdekoodi

Liite 17. IR-ohjainkortin lähettimen Interrupts.c-tiedoston lähdekoodi

Liite 18. IR-ohjainkortin lähettimen main.h-tiedoston lähdekoodi

Liite 19. IR-ohjainkortin lähettimen main.c-tiedoston lähdekoodi

1 Johdanto

Laitteiden etäsäätö automatisoidulla ja keskitetyllä ohjauksella on tapa, jolla ohjataan prosesseja suuressa teollisuudessa. Ennen transistorin keksimistä ja sitä seuraavan mikroprosessorin tuloa markkinoille teollisuuden prosessien ohjaus toteutettiin rele- ja elektroniputkitauluilla. Prosessorin tulo markkinoille muutti mittaus- ja säätötekniikan pisteeseen, jossa tapahtui siirtyminen kohti täysin automatisoituja järjestelmiä. Esimerkiksi käsikäyttöisen venttiilin avaaminen ja sulkeminen korvautui prosessoriohjatulla säätöjärjestelmällä, joka ottaa parametrinsa suoraan prosessin mittaus- ja valvontatiedoista.

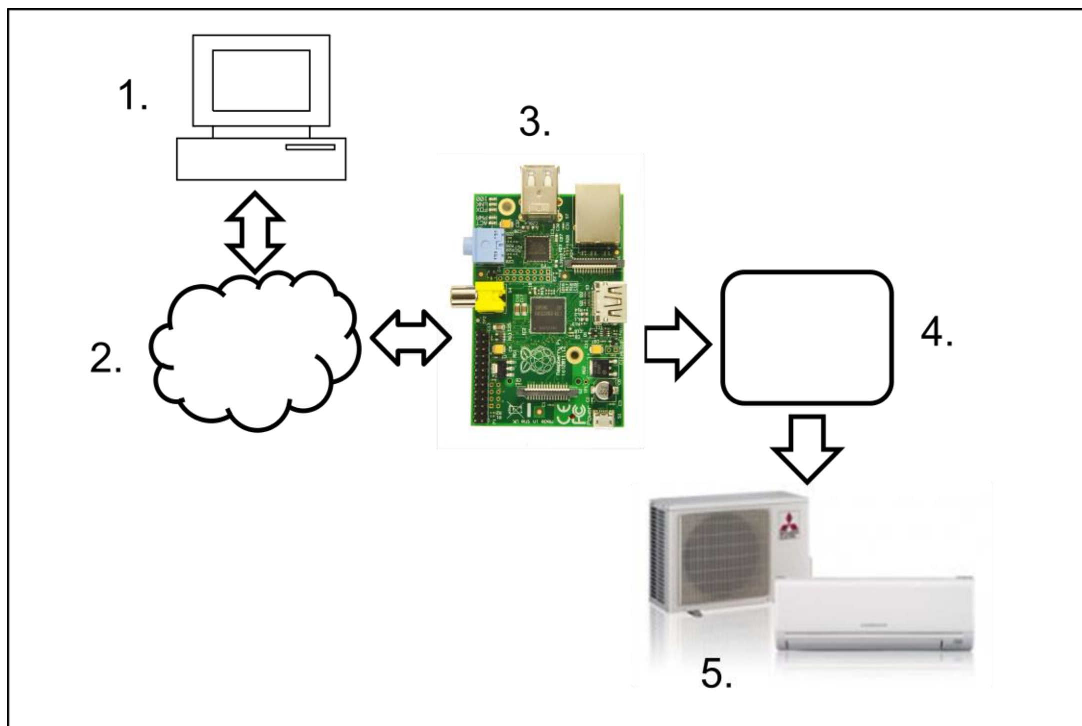
Etäsäätö ja -valvonta ei rajoitu pelkästään teollisuuden tarpeisiin, vaan tätä tekniikkaa on tuotu osaksi myös tavallisen ihmisen arkea. Hyvinä esimerkkeinä toimivat ”älykodit”. Konsepti pitää sisällään ajatuksen, jossa monet kodin toiminnot ovat automatisoituja sekä etähallittavissa. Esimerkkinä etähallinnasta voisi olla matkapuhelimien tekstiviestitoiminnoilla tapahtuva ohjaus, tai selainpohjainen yhteydenmuodostus kiinteistöjärjestelmään Internetin välityksellä. Vaikka ajatus automaattisesta kahvinkeittimestä, joka keittää kahvit valmiiksi ennen asukkaan kotiintuloa, kuulostaa sekä hauskalta että jotenkin tarpeettomalta, on ”älykodin” sovelluksilla kuitenkin paljon hyödyllisemmät tarkoitusperät: muun muassa asuntojen energiakustannusten hallinta automaattisella lämmönsäätelyllä, CO_2 -kaasujen mittaus ja ilmastoinnin säätely sekä se, että mahdollisen tulipalon sattuessa asunnossa toimiva järjestelmä itse kykenee ilmoittamaan hätäkeskukselle. Nämä muutamat esimerkit ovat vain suuntaa antavia sovelluksia suuresta kirjosta, joka määrittelee ”älykodin”. Elektronisten laitteiden laskeva hintakehitys sekä tekniikan kehittyessä nämä automaattiset valvonta- ja säätöratkaisut tulevat näkymään yhä useammassa kodissa.

Insinöörityön kohteena toimii kaukosäädinkäyttöinen ilmalämpöpumppu, jonka toimintaa pyritään vastaamaan ”älykodin” ominaispiirteitä. Tavoitteena on laajentaa ilmalämpöpumpun perinteistä kaukosäädinohjausta tuomalla mukaan etähallintamahdollisuus Internetin välityksellä. Tämä tulisi tapahtumaan rakentamalla kaukosäädintä mallintava IR-ohjainkortti, joka tullaan liittämään Linux-käyttöjärjestelmällä varustettuun tietokonekorttiin. Tietokonekortti tarjoaisi yksinkertaisen etähallintapalvelun ilmalämpöpumpun ohjaukseen. Tietokonekortin on tarkoitus samalla toimia laajennusalustana muille mahdollisille "älykodin" sovelluksille.

2 Kohdeympäristö, laitteet ja sovellukset

Kuvassa 1 on esitettynä periaatteellinen rakenne kohdeympäristöstä, jossa on tämän insinööriyön kannalta keskeisimmät komponentit. Kohdeympäristö tulisi koostumaan seuraavista osista:

1. Etäpääte kuvaa tavallista PC:tä, jolla voidaan ottaa yhteys tietokonekortissa toimivaan etäohjauspalvelimeen (3).
2. Internet- yhteys
3. Raspberry Pi [1], Linux -alusta, joka tulee toimimaan etähallintapalvelimena sekä alustana muille mahdollisille sovelluksille (mahdollisia laajennuksia ei tulla tässä työssä käsittelemään).
4. IR-ohjainkortti, laite mallintaa ilmalämpöpumpun omaa kaukosäädintä, joka vastaanottaa komentoja Linux -alustalta sekä muuntaa tiedon ilmalämpöpumpun ymmärtämään muotoon.
5. Mitsubishi MSZ-GE25VA ilmalämpöpumppu, johon toteutetaan etäohjaus Internetin välityksellä.



Kuva 1. Hahmotelma kohdejärjestelmästä.

Raspberry Pi

Raspberry Pi on edullinen yhdelle piirilevylle integroitu tietokone, joka on kehitetty Raspberry Pi -säätiön toimesta. Raspberry Pi -säätiön mukaan tämän pienen ja edullisen tietokoneen tarkoitusperät ovat edistää tietotekniikan opetusta kouluissa ja kolmansissa maissa. Raspberry Pi:n ensimmäinen erä tuli myyntiin helmikuussa 2012 ja on tämän opinnäytetyön kirjoitushetkeen mennessä kerännyt ympärilleen mittavan yhteisön. Internetissä on tarjolla Raspberry Pi:ta käsitteleviä verkkosivustoja, muun muassa Wikipedia sekä foorumisivusto. [2;3.]

Raspberry Pi:ta on insinöörityön kirjoitushetkellä tarjolla kahta mallia: A ja B. Taulukosta 1 voi havaita, että molemmat mallit ovat identtisiä monilla eri osa-alueilla. Poiketen A mallista B malli tarjoaa enemmän muistia, yhden ylimääräisen USB-liitännän sekä 10/100 Mbit/s Ethernet -portin.

Taulukko 1. Raspberry Pi:n tekniset tiedot. [4.]

	Malli A	**Malli B**
CPU	700 Mhz ARM11 (Broadcom BCM2835 SoC)	
GPU	Broadcom VideoCore IV (Broadcom BCM2835 SoC)	
Muisti (SDRAM)	256 MiB	512 MiB
USB2.0	1	2 (integroidulla hub:lla)
Videolähtö	Komposiitti video | RCA, HDMI (ei samanaikaisia)	
Audiolähtö	TRS | 3,5 mm liitäntä, HDMI	
Audiotulo	X	
Massamuistiliitäntä	SD/MMC/SDIO -korttipaikka	
Ethernet -liitäntä	X	10/100 RJ45
Oheislaitteet	GPIO, SPI, I2C I2S, UART	
Tehonkulutus	2,5 W	3,5 W
Virransyöttö	5 VDC microusb (tyyppi B) tai GPIO liitännän kautta	
Koko	85,60 mm x 53,98 mm	

Taulukon 1 kahdesta eri mallivaihtoehdosta A vaikuttaa otollisemmalta sen ~29 % pienemmän tehonkulutuksen vuoksi. Kuvan 1 mukaisen Internet-yhteysvaatimuksen seurauksena A-mallin puuttuva Ethernet-portti on rajoittava tekijä mallin valinnassa. Vaihtoehtoisesti A-mallin ainoa USB-paikka voisi toimia Wifi-adapterin liitäntänä. Toisesta näkökulmasta Wifi-adapterin omat virrankulutukset summattuna kyseinen ratkaisu todennäköisesti kumoaa A-mallin pienemmän tehonkulutuksesta saadun hyödyn. Kuitenkin esteettisestä näkökulmasta tämä ratkaisu voisi olla parempi, koska silloin ei olisi fyysistä kaapelia Internet-yhteyttä varten, mikä voisi haitata laitteen loppusijoitusta ajatellen. Kehitystyön näkökulmasta valitsin B mallin sen tarjoaman ylimääräisen USB- ja Ethernet -liitännän vuoksi.

Arch Linux ARM

Arch Linux on yksi lukuisista Linux-jakeluista, jonka pääkehittäjänä on vuoteen 2007 asti toiminut kanadalainen ohjelmoija Judd Vinet. Arch Linuxin ensimmäinen virallinen julkaisu tapahtui 11.3.2002, mikä tekee jakelusta suhteellisen nuoren verrattuna esimerkiksi Slackware Linuxiin, jonka ensimmäinen beeta-versio julkistettiin huhtikuussa vuonna 1993. [6;7.] Arch Linux noudattaa niin kutsuttua ”The Arch Way” -filosofiaa, jonka kulmakivet määrittävät Arch Linux jakelun yksinkertaiseksi, tietokoneen resursseja säästäväksi, joustavaksi sekä hyvin UNIX-tyyppiseksi. [8.]

Arch Linux ARM on oma haaransa Arch Linux -jakeluissa. Arch Linuxin pääpainon ollessa i686/x86_64 tietokonearkkitehtuureissa – Arch Linux ARM on käännetty ja optimoitu ARM -arkkitehtuurin ARMv5-7 -käskykannan omaaville prosessoreille. [5;6.] Arch Linux ARM tulee toimimaan Raspberry Pi:n käyttöjärjestelmänä.

IR-ohjainkortti

Monia ilmalämpöpumppuja, kuten tässä työssä kohdelaitteena olevaa Mitsubishi MSZ-GE25VA:ta ohjataan infrapunavalosignaalia lähettävällä kaukosäätimellä. IR-ohjainkortin tulee matkia kaukosäätimen toimintaa toimimalla infrapunalaitteena sekä tarjoamalla laiterajapinnan ulkoista ohjausta varten. IR-ohjainkortin suunnittelun lähtökohtana on toteuttaa prototyyppi, joka tulee ensisijaisesti toimimaan osana Raspberry PI:ta, mutta samalla huomioidaan, että IR-ohjainkortti olisi siirrettävissä alustalta toiselle; esimerkiksi tavalliseen pöytä PC:hen tai muuhun sulautettuun järjestelmään.

3 RS-232-sarjaportti

RS-232 (Recommended Standard 232) on nykyisellä nimellä toimiva Electronic Industries Associationin kehittämä asynkroninen sarjaportti kahden eri päätelaitteen välillä. Standardin ensimmäinen versio kehitettiin vuonna 1962, ja RS-232 -standardi määrittelee sähköiset ominaisuudet päätelaitteiston (DTE, Data Terminal Equipment) ja tietoliikennelaitteiston (DCE, Data Communication Equipment) väliltä. Tuolloin DCE esitti useimmissa tapauksissa modeemia, joka muunsi vastaanotetun datan muotoon, jolla data olisi mahdollista lähettää puhelinlinjoja pitkin. Vuodesta 1962 lähtien RS-232 -standardi on saanut useita nimiä ja revisioita. joista nykyisin standardi on EIA/TIA 232. [9.]

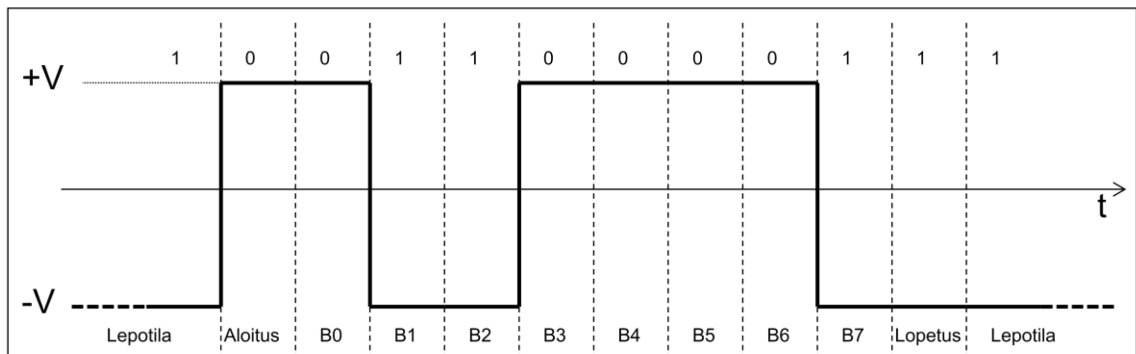
RS-232 on itsessään vanhentunutta teknologiaa ja USB (Universal Serial Bus) on korvannut RS-232:n oheislaiterajapintana. Tästä huolimatta RS-232:n yhteydessä käytetty universaali asynkroninen sarjaprotokolla on edelleen yleisesti käytetty muun muassa sulautetuissa järjestelmissä sen yksinkertaisuuden vuoksi.

Asynkronisen toiminnan seurauksena sarjaprotokollaa käytettävillä laitteilla täytyy olla keskinäinen sopimus tiedonsiirtonopeudesta, koska asynkronisesti toimivalla tiedonsiirtotavalla ei ole erillistä kellosignaalia, jolla voi tahdistaa laitteiden välisen tiedonsiirron. Molemmilla päätelaitteilla on oltava omat sisäiset kellonsa, joilla tahdistaa tiedon siirto- ja vastaanottotaajuus. [10; s. 11–12.] Tietoliikennetekniikassa tämä siirto- ja vastaanottotaajuus esitetään kahdella eri tavalla: bittinopeutena (bps) tai symbolinopeutena (baudi). Bittinopeudessa siirtonopeus esitetään niin, että ilmaistaan, kuinka monta bittiä siirtyy päätelaitteelta toiselle sekuntia kohden eli N bit/s. Symbolinopeudessa ilmaistaan signaalin muutosnopeus sekuntia kohden. Bittinopeudesta poiketen symbolinopeudessa yhdellä symbolilla voidaan esittää 1–N bittiä johtuen signaalin modulointimenetelmistä. Sarjamuotoisessa tiedonsiirrossa data lähetetään aina yksi bitti kerrallaan. Tuolloin voidaan ilmaista, että sarjamuotoisessa tiedonsiirtonopeudessa symbolinopeus on sama kuin bittinopeus. [10; s. 13–14.] Taulukossa 2 on esitettynä asynkronisen sarjaprotokollan tyypilliset tiedonsiirtonopeudet.

Taulukko 2. RS-232:n sarjaprotokollassa käytetyt tiedonsiirtonopeudet. [11.]

Bit/s	75	110	300	1200	2400	4800	9600	19200	38400	57600	115200

Tiedonsiirto asynkronisella sarjaprotokollalla tapahtuu lohkoissa, joita kutsutaan sanoiksi (word). Jokainen sana sisältää aloitusbitin, databitit, vaihtoehtoiset pariteettibitit sekä yhden tai useamman lopetusbitin. Yleisesti käytetty datan formaatti on muotoa 8-N-1, eli lähetettävässä sanassa on yksi aloitus- ja lopetusbitti, kahdeksan databittiä, eikä lainkaan pariteettibittiä. [10; s. 12–13.] Kuvassa 2 on esitettynä ajoituskaavio 8-N-1 formaatin ASCII-merkistön ”a” kirjaimen lähetysprosessi. Alkutilanteessa siirtokanava on lepotilassa, eli linjalla vaikuttaa -V-tason jännite. Seuraavaksi lähetin ilmaisee lähetysprosessin alkavan muuttamalla siirtokanavan jännitetason +V:n tasolle, eli linjalla vaikuttaa bittiarvo 0. Aloitusbittiä seuraa 8 databittiä, tässä siirtyy vastaanottimelle ”a”-kirjainta vastaava ASCII-bittijono. Lähetettävän lohkon lähetys päättyy lopetusbittiin, jonka jälkeen siirtokanava jää lepotilaan tai lähetin aloittaa seuraavan lohkon lähettämisen.



Kuva 2. Ajoituskaavio ASCII merkistön ”a” kirjaimen lähetyksestä 8-N-1 formaatilla.

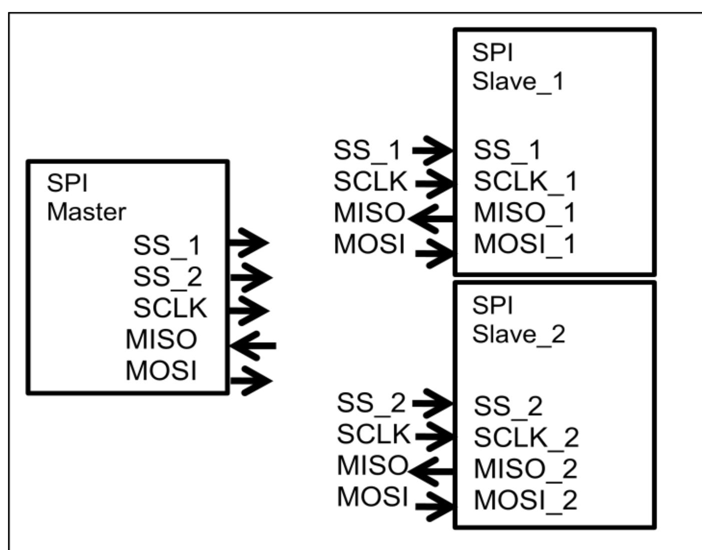
RS-232 -standardin laiterajapinta käyttää tiedonsiirrossa positiivisia ja negatiivisia jännitetasoja, jotka vaihtelevat väliltä -3 V – -15 V (looginen 1) ja +3 V – +15 V (looginen 0). [10; s. 46–47.] Yleisesti on käytetty ±12 V jännitetasoja, kuten esimerkiksi sarjaportilla varustetussa tietokoneessa.

Sulautetuissa järjestelmissä käytetty asynkroninen sarjaportti toteutetaan UART:lla (Universal Asynchronous/Receiver/Transmitter). Tämä piiri löytyy integroituna miltei jokaisesta mikrokontrollerista. UART -piiri toimii samoilla käyttöjännitteillä kuin mikrokontrollerit, ja käyttämällä ±12 V jännitetasoja seuraa piirin rikkoutuminen. UART:ssa RS-232:een yhteensopivat jännitetasot toteutetaan erillisellä muuntopiirillä, kuten esimerkiksi Maxim Integrated -yhtiön valmistamalla MAX232-muuntopiirillä, joka muuttaa UART-yhteensopivat jännitetasot RS-232-yhteensopiviksi ja toisinpäin.

RS-232-laiterajapinnassa data siirretään käyttämällä lähetys ja vastaanottolinjoja (rx, tx). Vaihtoehtoisesti on käytössä myös erilaisia oheislinjoja, joilla hallitaan datansiirtoa, mutta niitä ei käsitellä. Käyttämällä rx:ää, tx:ää ja yhteisen maatason linjoja on käytössä kytkentä, jota kutsutaan nimellä ”null modem” -kytkennäksi. "Null modem" -kytkentä on kaikkein yleisin ja yksinkertaisin tapa siirtää dataa kahden eri päätelaitteen väliltä.

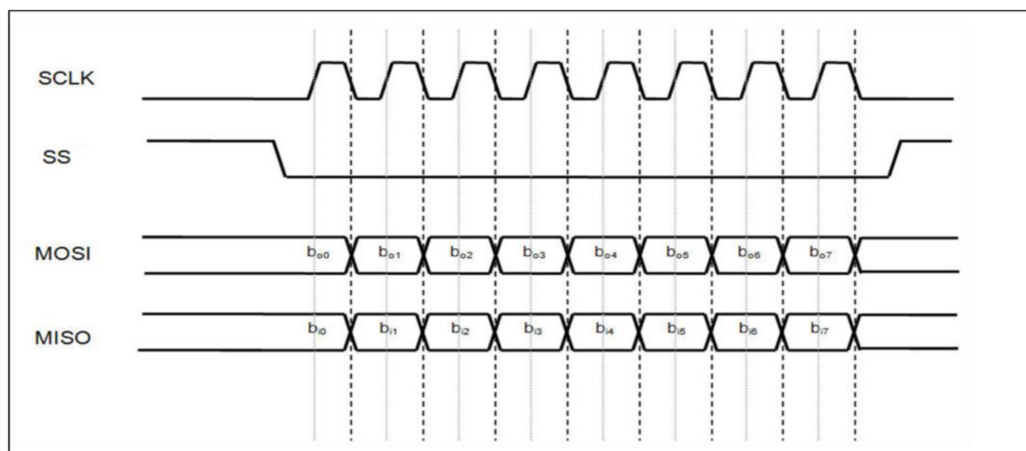
4 SPI-protokolla

SPI (Serial Peripheral Interface) on Motorolan kehittämä synkroninen sarjamuotoinen tiedonsiirtoprotokolla. SPI-väylää käyttävät laitteet toimivat isäntä-oheislaiteperiaatteella, jossa väylässä on yksi isäntälaite, sekä 1–N kappaletta oheislaitteita. SPI operoi full-duplex-moodissa, eli väylässä olevat laitteet lähettävät ja vastaanottavat dataa samanaikaisesti. Kahden laitteen väylässä SPI rakentuu neljästä signaalista kello-, lähetys-, vastaanotto- sekä valintasignaalista. Kellosignaali (SCLK) toimii datan siirron tahdistimena, jolla ilmaistaan hetkeä milloin bitti vastaanotetaan väylää käyttävältä laitteelta toiselle. MOSI (master out, slave in) ja MISO (master in, slave out) -signaaleja pitkin liikkuu isäntä- ja oheislaitteen välinen data. Valintasignaalilla (SS) isäntälaite osoittaa oheislaitetta, jonka kanssa datan siirto tulee tapahtumaan. Lisäämällä useampia valintasignaaleja voidaan samaan SPI-väylään liittää useampia oheislaitteita. Ainoana oheislaitteiden määrää rajoittavana tekijänä on isäntälaitteen fyysisten valintasignaalien lukumäärä. Esimerkkikytkentä yhden isäntälaitteen ja kahden oheislaitteen systeemistä on esitettynä kuvassa 3.



Kuva 3. Esimerkkiesitys kolmen laitteen SPI-väylästä.

SPI-väylää käyttävien laitteiden välinen datan siirto tapahtuu kehyksissä (frame), joiden konkreettinen rakenne vaihtelee oheislaitteen laitevalmistajan spesifikaatioiden ja suunnitteluperiaatteiden mukaan. Lähtökohtaisesti voidaan olettaa, että jokainen kehys alkaa ja päättyy valintasignaalin (SS) tilanmuutokseen. Valintasignaalin tilanmuutoksen jälkeen seuraa N-bittiä laitteiden väliltä siirrettävää dataa, joista jokainen bitti tahdistetaan kellosignaalilla. Yhtenä esimerkkinä on kuvassa 4 oleva ajoituskaavio. Kuvassa kehys alkaa valintasignaalin siirtyessä korkeammasta jännitetasosta matalampaan jännitetasoon. Kehyksen jokainen lähetettävä bitti tahdistetaan kellosignaalin nousevalla reunalla, kunnes kehyksen kaikki bitit on siirretty laitteelta toiselle. Kehyksen päättyminen ilmaistaan nostamalla valintasignaali takaisin korkeampaan jännitetasoon.



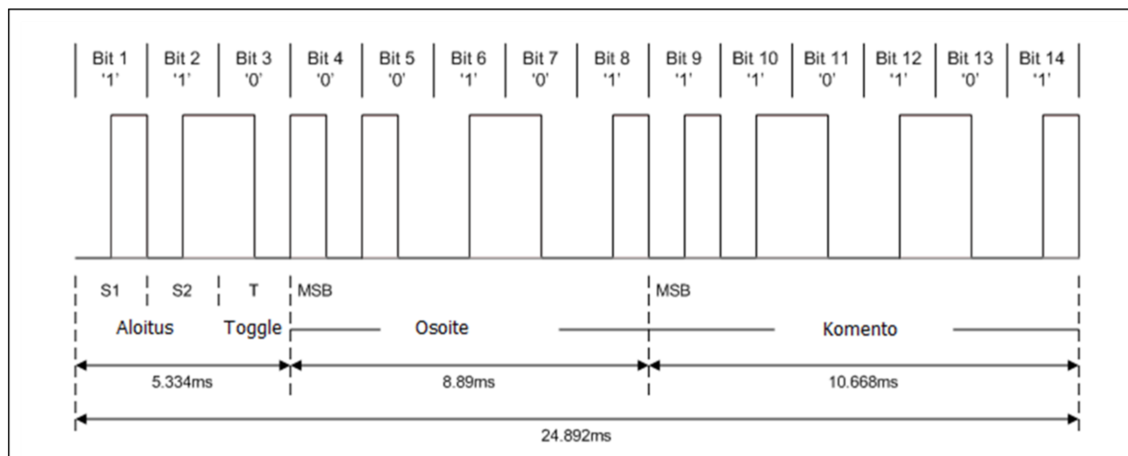
Kuva 4. Esimerkki SPI-väylän ajoituskaaviosta.

5 Optinen tiedonsiirto

Optisella tiedonsiirrolla tarkoitetaan menetelmiä siirtää informaatiota vapaassa tilassa useimmissa tapauksissa infrapunavaloa tai ultraviolettivaloa hyödyntäen. Tyypillisesti kaksipiste-periaatteella toimiva tiedonsiirto koostuu lähettimestä sekä vastaanottimesta. Vastaanotin koostuu valoherkästä komponentista sekä muuntopiiristä, joka muuntaa vastaanotetun valosignaalin sähköiseen muotoon. Valosignaalin vastaanotinpää voi olla sijoitettuna optisen suodattimen taakse, jolloin valoherkkään komponenttiin kohdistuu vain tietyt valon spektrin aallonpituudet.

Kuluttajaelektroniikassa audio- ja videolaitteiden sekä kaukosäätimen välinen tiedonsiirto monissa tapauksissa on toteutettu käyttäen infrapunavaloa. Käytetty infrapunavalo vaikuttaa 870, 930–950 nanometrin aallonpituuksilla ja kuuluu near-infrared (NIR)-kategoriaan. Informaation siirto kaukosäätimen ja vastelaitteen väliltä toteutuu käyttä-

mällä optiseen tiedonsiirtoon soveltuvaa sarjamuotoista tiedonsiirtoprotokollaa tai standardia. Käyttämällä tiettyjä protokollia tai standardeja kuluttajaelektroniikan laitevalmistajat varmistavat, että samantyyppiset infrapunavaloa käyttävät laitteet eivät erehdy vastaanottamaan niille kuulumatonta informaatiota. Protokollasta tai standardista riippuen informaation siirto tapahtuu kehyksissä (frame), johon tyypillisesti pakataan ajastus, osoite, data, sekä aloitus sekä lopetusbitit. Kehys lähetetään moduloimalla infrapunavalolähdettä käytetystä protokollasta riippuen 30–60 kHz:n kantoaallolla parantamaan vastaanottimen häiriönsietoa. [13.] Esimerkkinä olkoon Philips-yhtiön kehittämä RC-5-protokolla, jonka kehys on esitettynä kuvassa 5.



Kuva 5. Philips RC-5-protokollan kehys. [14.]

RC-5-protokollassa bittien koodaukseen käytetään Manchester-koodausmenetelmää, jossa looginen 0 esitetään jännitetason siirtymätilaa korkeasta tasosta matalaan tasoon ja looginen 1 matalasta tasosta korkeaan tasoon. Jokainen bitti siirretään 36 kHz:n kantoaallolla, jonka seurauksena kehyksen jokaisen loogisen tilan esityksen kesto on 1,778 millisekuntia. Bitti jaetaan kestoltaan kahteen yhtä pitkään jaksoon: space ja burst. Space kuvaa tilaa, jossa kyseisellä ajanhetkellä kuvataan vastaanottimelle matalaa jännitetasoa. Vastaavasti burst-tila kuvaa korkeaa jännitetasoa. On tavanomaista, että burst -tilan aikana lähetin suorittaa pulssileveysmodulaatiota pulssisuhteella 1/3 tai 1/4.

Lähetettävä kehys koostuu 14 bitistä. Ensimmäiset kaksi bittiä (S1, S2) ilmaisevat kehyksen aloitusta. Toggle-bitillä ilmaistaan esimerkiksi kaukosäätimen nappien painelujen tilaa: onko nappia painettu kerran vai yhtäjaksoisesti. Toggle-bittiä seuraa viisi osoitebittiä, joilla vastaanotin selvittää itselleen kuuluvan kehyksen. Viimeiset kuusi komen-

tobittiä ilmaisevat komennon tai komentojonon, jota vastaanotettava laite suorittaa. Kokonaisen kehyksen lähettämiseen ja vastaanottamiseen kuluu vajaat 25 ms.

RC-5 on yksi lukuisista kulutuselektroniikan protokollista. Lukuisien protokollien lisäksi päätteiden ohjaukseen lähetettävät kehykset ovat laitekohtaisia, mikä johtaa tilanteeseen, jossa jokaista viihde-elektroniikan laitetta kohden on oma kaukosäätimensä. Lähtökohtaisesti voi pitää, että jokaista suurta kulutuselektroniikan valmistajaa kohden on kehitettynä kaukosäätimiä varten oma infrapunamuotoiseen tiedonsiirtoon tarkoitettu protokolla.

6 Linux-laiteajurit

6.1 Linux

Linux Linus Torvaldsin kehittämä avoimeen lähdekoodiin perustuva käyttöjärjestelmäydin, joka on lisensoitu GPLv2-lisenssillä. Linux ydin, eli kernel on GNU-Linux käyttöjärjestelmän alin osa, joka hallinnoi ja mahdollistaa kaikkien muiden tietokoneen ohjelmien toiminnan. Ensimmäinen Linuxin versio julkaistiin vuonna 1991 ja tähän päivään mennessä Linuxin lähdekoodipuu sisältää yli 15 miljoonaa riviä ohjelmakoodia. [15.]

Linux on monoliittinen ydin, eli Linux on isohko suoritettava ohjelma, jossa kaikki Linuxin osat toimivat samassa muistiavaruudessa. Ytimen päätoiminnot jaetaan viiteen eri osaan: prosessihallintaan, muistinhallintaan, tiedostojärjestelmään, laitehallintaan sekä verkkotoimintoihin.

Prosessinhallinnan tehtäviä ovat sovellusten prosessien luonti ja tuhoaminen sekä prosessien ja I/O laitteiden välisen kommunikoinnin hallinta. Vuorottaja (scheduler), joka on osana prosessihallintaa, hallinnoi suorittavien sovellusten tai prosessien kontekstinvaihtoja. Vuorottajalla aikaansaadaan järjestelmän näennäisrinnakkaisuus huolehtimalla, että jokainen prosessi saa riittävästi prosessoriaikaa.

Muistinhallinta vastaa järjestelmän muistin varaamisesta ja vapauttamisesta. Linuxissa muistinhallinta luo kaikille prosesseille virtuaalisen muistiavaruuden, joka käännetään prosessorin MMU-yksikön (Memory Management Unit) avulla fyysisiksi osoitteiksi.

Tiedostojärjestelmä rakentaa strukturoituja tiedostojärjestelmiä fyysisten medioiden päälle. Linux tukee erilaisia tiedostojärjestelmiä, ja monet asiat Linuxissa esitetään tiedostoina. Tämä ominaisuus on lainattu alkujaan Unix-järjestelmästä.

Laitehallinta koostuu laiteajureista, jotka toimivat liityntäkohtina kaikkiin Linux-järjestelmän fyysisiin laitteisiin. Jotta laitetta olisi mahdollista käyttää, täytyy ytimessä olla laiteajuri toteutettuna jokaista järjestelmässä sijaitsevaa laitetyyppiä kohden.

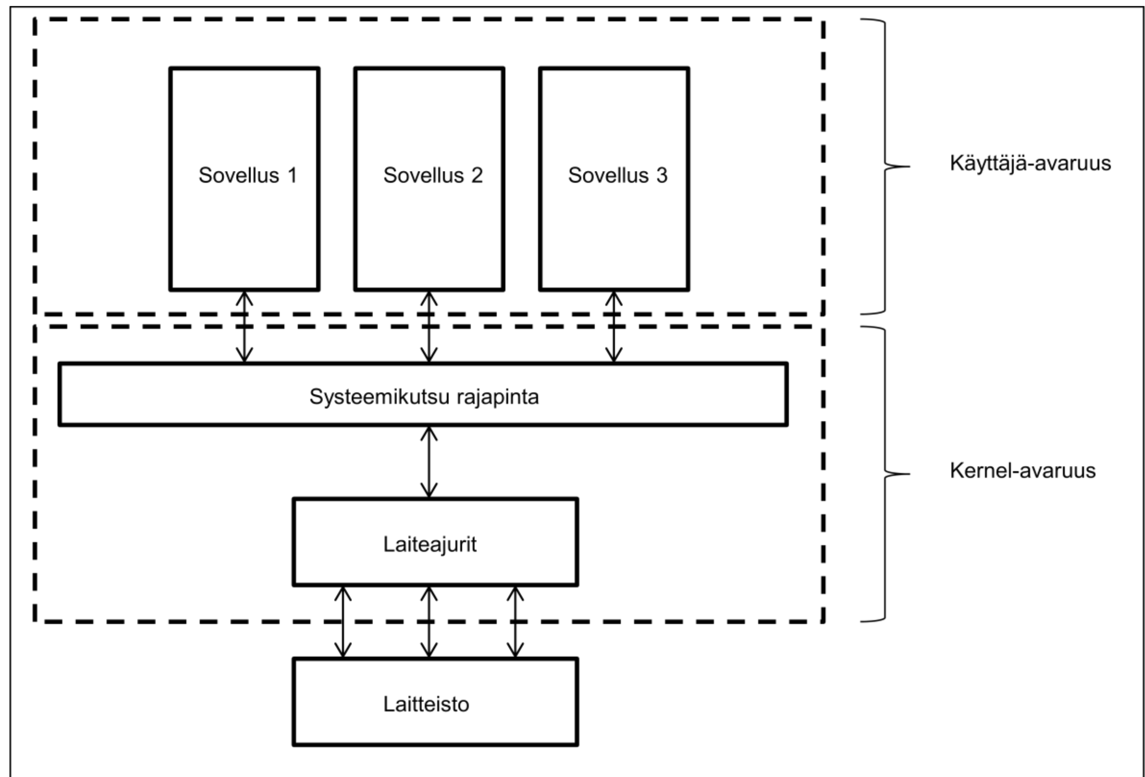
Verkkotoiminnot vastaanottavat, identifioivat sekä lähettävät verkkopaketteja. Järjestelmä vastaa verkkopakettien jakelusta prosessien ja verkkokorttien kesken sekä kontrolloi ohjelmien suoritusta pohjautuen niiden verkkotoimintoihin. [16, s. 4–5.]

Linux-ydin toimii etuoikeutetussa tilassa, eli ytimen osilla on täydet oikeudet hallita järjestelmän resursseja, tällöin puhutaan kernel-avaruuden (kernel space) suoritustasosta. Vastaavasti Linuxissa on olemassa alempi taso, jota kutsutaan käyttäjäavaruuden (user space) tasoksi. Käyttäjäavaruudessa suoritetaan Linuxissa toimivat käyttäjän sovellukset ja prosessit. Käyttäjäavaruudessa suoritettavilla sovelluksilla pääsy ytimen muistiavaruuteen on estetty, mutta Linux-ydin tarjoaa rajapinnan, jolla sovellukset voivat kommunikoida ytimen osien kanssa. Keskustelu ytimen ja sovellusten kesken käydään systeemikutsujen (System call) kautta. Systeemikutsuja tehdään sovelluskirjastojen avulla, jotka välittävät systeemikutsun parametrit systeemikutsurajapinnalle. Systeemikutsufunktiot suoritetaan Linuxin kernel-avaruudessa. [17, s. 4–5].

6.2 Laiteajurien tehtävä

Linux -laiteajurit eli moduulit ovat tarkoin määriteltyjä ohjelmia, jotka toimivat Linux-ytimen laajennusosina ja sijaitsevat kernel-avaruudessa. Moduulit nimetään laiteajureiksi, mikä johtuu niiden erityisroolista Linuxissa. Suurin osa moduuleista toimii Linux-järjestelmään liitettyjen laitteiden ohjaimina. Laiteajureiden tarkoitus on esiintyä ”mustina laatikkoina” toteuttamalla rajapinta jolla voidaan Linux järjestelmään liitetyn laitteen vastaamaan järjestelmän tai käyttäjän pyyntöihin. Mustan laatikon periaatteella on tarkoitus peittää yksityiskohdat laitetason toiminnasta sekä tuottaa käyttäjäavaruuden sovellukselle liityntäkohta, jolla mahdollistetaan keskustelu laitteen ja järjestelmän/käyttäjän välillä.

Kuvassa 6 on esitettynä, kuinka käyttäjäavaruudessa toimivat sovellukset keskustelevat laiteajureiden kanssa. Riippuen toteutuksesta laiteajuri tarjoaa liityntäkohdat, joihin sovellukset "kiinnittyvät" Linuxin systeemikutsurajapinnan kautta. [18, s. 16.]



Kuva 6. Sovellusten liityntä laiteajureihin.

Laiteajurit jaetaan käyttötarkoituksensa mukaan kolmeen pääryhmään: karakteri- (character), lohko- (block) ja verkkolaitteisiin (network).

Karakterilaitteet esittävät laitteita, joita käsitellään datavirtana. Laitteisiin kirjoitetaan ja laitteista luetaan dataa tavu kerrallaan. Esimerkkejä karakterilaitteista ovat sarjaportit, näytöt ja näppäimistöt. Karakterilaitetta mallintava laiteajuri toteuttaa vähintään *open-, close-, read-* ja *write*-funktiot.

Lohkolaitteet esittävät laitteita, joihin kirjoittaminen ja lukeminen kohdistuvat lohkossa tai lohkon moninkerroissa. Linux liittää tiedostojärjestelmät lohkolaitteiden päälle, jolloin laitteen sisältämä data esitetään strukturoidussa muodossa. Massamuistilaitteet ovat esimerkkejä lohkolaitteista.

Verkkolaitteet muistuttavat toiminnaltaan karakterilaitteita, mutta data lähetetään tavallisen datavirran sijaan pakettien muodossa. Karakterilaitteiden *read-* ja *write* -funktioiden sijasta kernel kutsuu verkkolaitetyyppisen ajurin pakettien lähetys- ja vastaanottofunktiot. [18, s. 11–12.]

6.3 Laiteajurin runko

Laiteajurien käytännön toteutukset hieman poikkeavat tavallisesta käyttäjäavaruuden sovelluksista. Laiteajurit toteutetaan monien käyttäjäavaruuden sovellusten lailla C-kielellä, erona on menetelmä, kuinka laiteajurin ja tavallisen sovelluksen elinkaari alkaa ja päättyy. Käyttäjäavaruuden ohjelman suoritus alkaa *main*-funktiosta. Laiteajuri puolestaan aloittaa toimintansa liittyessään Linux-ytimeen. Ajurin liittyessä osaksi ydintä, suoritetaan ajurin *init*-takaisinkutsufunktio. *Init*-takaisinkutsufunktio muistuttaa toiminnaltaan tavallisen sovelluksen *main*-funktiota, mutta funktio suoritetaan tasan kerran sekä suorituksen jälkeen init-funktion ohjelmakoodi poistetaan kokonaan muistista. Vastaavasti laiteajurin suoritus loppuu *exit*-takaisinkutsufunktioon. Ydin suorittaa *exit*-funktion silloin kun laiteajuri poistetaan ytimestä. *Exit*-funktion tarkoitus on toimia siivousfunktiona, jossa kumotaan *init*-funktiossa tehdyt asiat, kuten resurssien ja varatun muistin vapautus. *init-* ja *exit*-takaisinkutsufunktiot rekisteröidään kernel API:a vasten, käyttämällä ohjelmakoodissa *module_init-* ja *module_exit* -makroja. Esimerkki laiteajurin rungosta on esitettynä ohjelmalistauksessa 1.

Linux-laiteajurien lähdekoodit sisältävät muutamia informaatiomakroja, jotka käsittelevät ajurin lisensiointia, sekä muuta ajuria koskevaa informaatiota. Yleisimmät käytetyt informaatiomakrot ovat *MODULE_AUTHOR, MODULE_DESCRIPTION, MODULE_LICENSE, MODULE_VERSION*. Nämä makrot eivät suoranaisesti vaikuta laiteajurin toimintaan, vaan niiden tarkoitus on antaa käyttäjälle tietoa ajurista *modinfo*-sovelluksen avulla. Poikkeuksena muista informaatiomakroista on *MODULE_LICENSE*, jonka poisjättämisen ydin tulkitsee automaattisesti suljetun lähdekoodin ajuriksi ja tuottaa varoituksen ajurin latauksen yhteydessä. [18, s. 24–28;19, s. 5–8.]

```
#include <linux/module.h> /*käytettävä kaikissa ajureissa*/
#include <linux/init.h> /*sisältää module_init ja -exit makrot*/

static int __init ajurin_init(void) /*init takaisinkutsufunktio*/
{
    /* toteutus*/
    return 0;
}

static void __exit ajurin_exit(void) /*exit takaisinkutsufunktio*/
{
    /*toteutus*/
}

module_init(ajurin_init);/*init rekisteröintimakro*/
module_exit(ajurin_exit);/*exit rekisteröintimakro*/

MODULE_LICENSE("GPL");/*ajurin lisenssi*/
MODULE_AUTHOR("Vladimir Byckling");/*tekijän nimi*/
MODULE_DESCRIPTION("laiterunko");/*ajurin selite*/
MODULE_VERSION("0.1");/*ajurin versio*/
```

Ohjelmalistaus 1. Esitys laiteajurin rungosta.

6.4 Laiteajurin kääntäminen

Laiteajurit käännetään tarkoituksen mukaan joko staattisesti ytimeen tai moduuleiksi. Staattisessa menetelmässä ajurin lähdekoodi otetaan mukaan itse ytimen käännösprosessiin, jolloin Linux-ytimen mukana käännetty ajuri on aina osana ydintä. Moduuleiksi käännettyjen ajureiden lopputuloksena on binäärinen ko-päätteinen tiedosto. Moduuleiksi käännettävät ajurit käännetään Linuxin lähdekoodipuuta vasten, jossa on suoritettu ytimen käännösvaihe, sekä ytimelle on moduulien liittäminen sallittuna ominaisuutena. [18, s. 28–29;19, s. 6–7.]

Vastaavasti, jos käytössä on jokin Linux-jakelu, eikä käytössä olevan ytimen käännettyä lähdekoodipuuta ole helposti saatavilla, niin silloin ladataan kyseisen Linux-jakelun ylläpitäjän tarjoamat ”Linux headers” -tiedostot. ”Linux headers” on riisuttu Linux-lähdekoodipuu, josta on poistettu kaikki muu paitsi tarpeelliset tiedostot, joita tarvitaan laiteajureiden käännösoperaatiossa. Suurimmaksi osaksi ”Linux headers” sisältävät tarvittavat skriptit ja C/C++ -kielen h-pääteiset headers-tiedostot, jotka toimivat Linux-ytimen API-tiedostoina.

Linuxin 2.6 versiosta lähtien lähdekoodipuu käyttää käännösprosesseihin kbuild -työkalua. Käännösohjeet luodaan kbuild-tekstitiedostoilla, joita myös kutsutaan nimellä *Makefile.* Käännettävää ajuria varten *Makefile* voidaan toteuttaa ohjelmalistaus 2 mukaisilla esimerkkisäännöillä.

```
obj-m := ajuri.o
DIR   := /lib/modules/$(shell uname -r)/build
WD    := $(shell pwd)
default:
    $(MAKE) -C $(DIR) M=$(WD) modules;
```

Ohjelmalistaus 2. Makefile:n toteutus Linux moduulin kääntämistä varten.

Ohjelmalistauksessa 2 olevassa Makefilessa kbuildille välitetään tiedot, jossa ajuri.c-nimistä lähdekooditiedostoa ollaan kääntämässä moduuliksi (obj-m := ajuri.o). DIR ja WD esittävät apumuuttujia, joilla viitataan hakemistopolkuihin. DIR apumuuttuja esittää hakemistopolkua, jossa sijaitsee tarvittava Linux-lähdekoodipuu tai ”Linux headers” tiedostot, joita vasten moduuli käännetään. WD-apumuuttuja esittää hakemistopolkua, josta tarvittavaa lähdekooditiedostoa etsitään. Viimeisellä rivillä suoritetaan käännöskomento. Komennossa kutsutaan kbuild -työkalua, joka ohjaa ajuri.c lähdekooditiedoston käännösoperaatiota ajuri.ko -moduuliksi. Ohjelmalistaus 2 esimerkin mukainen laiteajurin kääntäminen aloitetaan siirtymällä konsolissa hakemistopolkuun, jossa ohjelmalistaus 2:n mukainen Makefile-tiedosto sijaitsee ja suoritetaan Linuxin konsolissa make-komento. [18, s. 28–29.]

6.5 Laiteajurin liittäminen ytimeen

Laiteajurit (moduulit) liitetään Linux-ytimeen staattisesti tai dynaamisesti. Luvussa 6.4 oli mainittuna, että staattisesti liitetyt moduulit liitetään ytimeen käännösprosessin aikana, jolloin moduuli on aina osana ydintä. Dynaamisessa menetelmässä ko-päätteinen moduuli liitetään ytimeen ajon aikana käyttämällä *module-init-tools* -paketin *insmod-* ja *rmmod-*, tai *modprobe*-sovelluksia. *Insmod* lataa laiteajurin ytimeen muun muassa selvittämällä kaikki tarvittavat ytimen symbolit symbolitaulusta ja liittämällä ajuri osaksi ytimen muistiavaruutta. Laiteajuri liitetään ytimeen käyttämällä komentoa *insmod ajuri.ko*. Vastaavasti ajuri poistetaan ytimestä käyttämällä komentoa *rmmod ajuri.* Laiteajuria poistettaessa ytimestä käytetään ainoastaan laiteajurin nimeä ilman ko-päätettä. *Modprobe* on monipuolisempi työkalu moduulien käsittelyyn. *Modprobe* kykenee selvit-

tämään laiteajurien keskinäiset riippuvuudet ja tarvittaessa lataamaan ytimeen useampia toisistaan riippuvia ajureita. Laiteajuri liitetään ytimeen käyttämällä komentoa *modprobe ajuri.* Vastaavasti laiteajuri poistetaan ytimestä käyttämällä komentoa *modprobe –r ajuri.* [17, s. 343–344]

6.6 Laiteajurit ja käyttäjäavaruus

Käyttäjäavaruuden sovelluksen tai prosessin tarve kommunikoida laiteajurin kanssa ei onnistu suoraan, koska käyttäjäavaruuden sovelluksilla ei ole suoraa pääsyä kernel-avaruuteen. Jotta käyttäjäavaruuden sovellukset voisivat kommunikoida laiteajurin kanssa, on laiteajurien tarjottava käyttötarkoituksesta riippuvia yhden tai useampia liityntärajapintoja, jotka ovat liitoksissa systeemikutsurajapintaan (Ks. luku 6.2, kuva 6). Seuraavaksi esitetään yleisellä tasolla liityntärajapintoja, joiden avulla käyttäjäavaruuden sovellukset voivat keskustella laiteajureiden kanssa.

6.6.1 Laitetiedostot

Laitetiedosto (Device node) on erikoisluontoinen tiedosto, joka sijaitsee oletusarvoisesti Linuxin */dev* hakemistossa. Laitetiedostot ovat kytköksissä laiteajureihin, jotka ydin yhdistää toisiinsa *major-* ja *minor*-luvuilla. *Major*-luku ilmaisee, mitä ajuria käytetään, kun keskustellaan laitteen kanssa, eli kaikki laitteet, joilla on sama *major*-luku käyttävät samaa ajuria. *Minor*-luvulla erotetaan kaikki laitteet, joita ajuri hallitsee. Esimerkkinä on kuvassa 7 massamuistilaitetta esittävä mmcblk0, joka on osioitu kahteen osaan p1 ja p2. Kaikilla laitetiedostoilla on sama *major*-luku 179, sekä *minor*-luvut 0–2, jotka erottavat laitetiedostot toisistaan. [18, s. 36–42.]

```
brw-rw---- 1 root disk    179,   0 Jun 19 11:49 mmcblk0
brw-rw---- 1 root disk    179,   1 Jun 19 11:49 mmcblk0p1
brw-rw---- 1 root disk    179,   2 Jun 19 11:49 mmcblk0p2
crw-rw---- 1 root dialout   4,  64 Jun 19 11:49 ttyS0
crw-rw---- 1 root dialout   4,  65 Jun 19 11:49 ttyS1
```

Kuva 7. Esimerkki /dev hakemistossa olevista laitetiedostoista.

Linuxin versiosta 2.6 lähtien laitetiedostoja käyttävät laiteajurit luovat laitetiedoston käyttämällä udev-rajapintaa. Linuxin 2.4 sekä aiempien versioiden laitetiedostot jouduttiin luomaan manuaalisesti, ja se johti tilanteeseen, jossa Linux jakeluiden mukana tuli suuri määrä laitetiedostoja, joilla ei välttämättä ollut käyttöä. Udev-rajapinta on kehitetty vastaamaan laitetiedostoista luomalla ja poistamalla ne tarpeen mukaan.

Laitetiedostot keskustelevat laiteajureiden kanssa käyttämällä standardeja input/output-systeemikutsuja. Laiteajuri toteuttaa käytettävät toiminnot käyttämällä Linux API fs.h-tiedostossa määriteltyä ohjelmalistauksessa 3 mukaista *file_operations* -tietorakennetta.

```
struct file_operations fops = {
    .owner    = THIS_MODULE,
    .open     = ajuri_open,
    .release = ajuri_close,
    .read     = ajuri_read,
    .write    = ajuri_write,
    .unlocked_ioctl    = ajuri_ioctl,
    /*…*/
};

struct module *owner;
int (*open) (struct inode *, struct file *);
int (*release) (struct inode *, struct file *);
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
```

Ohjelmalistaus 3. File_operations-tietorakenne ja funktio-osoittimien prototyypit.

File_operations -tietorakenteen kentät ovat funktio-osoittimia Ohjelmalistaus 3 esiintyviin funktion prototyyppeihin. Funktio-osoittimiin sijoitetaan ajurin osoitinta vastaava toteutus, näin ollen laitetiedostoihin kohdistuvat operaatiot ohjautuvat systeemikutsujen kautta oikeaan funktion toteutukseen. [18, s. 42–43.]

6.6.2 Proc

Proc on pseudo-tiedostojärjestelmä, joka sijaitsee keskusmuistissa. Proc liitetään fyysisen tiedostojärjestelmän */proc*-hakemistoon ja tavanomaisesti pitää sisällään Linux-järjestelmän toimintaa ja prosesseja koskevaa tietoa. Proc:n sisältämät tietueet esitetään ja käsitellään tekstitiedostoina ja laiteajureiden liittyminen Proc-

tiedostojärjestelmään ei ole poikkeus. Proc:ssa sijaitsevat laiteajureiden tietueet sijoitetaan tavallisesti /proc/drivers -hakemiston alle. Proc:n käyttö laiteajureissa toteutetaan käyttämällä Linux API:n proc_fs.h tiedostossa määriteltyä ohjelmalistauksessa 4 mukaista *proc_dir_entry* -tietorakennetta ja funktioita. [18, s. 161–162]

```
struct proc_dir_entry {
        read_proc_t *read_proc;
        write_proc_t *write_proc;
       /*...*/
};

struct proc_dir_entry *create_proc_entry
        (const char *name, mode_t mode, struct proc_dir_entry *parent);
void remove_proc_entry
        (const char *name, struct proc_dir_entry *parent);
int ajuri_lue
        (char *page, char **start, off_t off, int count, int *eof, void*data);
int ajuri_kirjoitus
        (struct file *file, const char __user * buffer, unsigned long count, void *data);
```

Ohjelmalistaus 4. Proc_dir_entry -rakenteen osa ja käytettyjen funktioiden prototyypit.

Ohjelmalistaus 4 esittämä *proc_dir_entry* -tietorakenne sisältää useita kenttiä, mutta esille on ainoastaan otettu kaksi tärkeää funktio-osoitinta: *read_proc* ja *write_proc*. Näitä takaisinkutsufunktiota suoritetaan, kun laiteajurin luomaan tiedostoon suoritetaan luku- tai kirjoitusoperaatiota. Funktio-osoittimeen *read_proc* viitataan toteuttamalla ohjelmalistaus 4 *ajuri_lue* -funktion prototyypin mukaisella toteutuksella, jota kutsutaan tiedoston lukuoperaation yhteydessä. Vastaavasti *write_proc*-takaisinkutsufunktioon viitataan toteuttamalla *ajuri_kirjoitus* prototyypin mukaisella toteutuksella. Proc-tietue luodaan ohjelmalistaus 4 funktiolla *create_proc_entry*, joka luo tietueen sekä palauttaa osoittimen *proc_dir_entry*-tietorakenteeseen. Laiteajurin elinkaaren lopussa proc:ssa sijaitseva tietue poistetaan käyttämällä funktiota *remove_proc_entry*. [18, s. 162–164.]

6.6.3 Unified Device Model ja Sysfs

Unified Device Model on rajapinta, joka on otettiin käyttöön Linuxin versiosta 2.6 lähtien. Unified Device Model tuottaa mekanismin järjestelmässä sijaitsevien laitteiden esittämiseen sekä kuvaamalla laitteiden topologian. Unified Device Modelin kehittämisen tuloksena sen ominaisuuksiin lukeutui laiteajureiden lähdekoodin osien monistamisen väheneminen, kyky listata kaikki järjestelmään liittyneet laitteet sekä nähdä eri laitteiden status ja mihin väylään laitteet ovat liittyneet. Tämän lisäksi Unified Device Modelin

yksi tärkeä kulmakivi on toteuttaa laitetason virranhallintaominaisuus Linuxin ytimessä. [17, s. 348–349]

Unified Device Modelin mukainen laiteajurin toteutus rakentuu kobject-tietorakenteen ympärille. Kobject esiintyy abstraktina tietorakenteena, joka olio-ohjelmoinnin termein ”peritään” laiteajurin lähdekoodissa laitetta kuvaavaan tietorakenteeseen. Jokaista fyysistä laitetta kohden esiintyy yleisluontoinen *device*-tietorakenne, joka on määritelty Linux API:n linux/device.h-tiedostossa. Tämä *device*-tietorakenne pitää sisällään datakenttiä, joilla pystytään kuvaamaan jokainen järjestelmään liitetty fyysinen laite sekä laitteeseen yhdistetty ajuri. *Device*-tietorakenne on esitettynä ohjelmalistauksessa 5. [18, s. 171–173.]

```
struct device {
        struct device *parent;
        struct device_private *p;
        struct kobject kobj;
        const char *init_name;          /* initial name of the device */
        const struct device_type *type;
        struct mutex mutex;             /* mutex to synchronize calls to its driver.*/
        struct bus_type *bus;           /* type of bus device is on */
        struct device_driver *driver;   /* which driver has allocated this  device */
        void    *platform_data;         /* Platform specific data, device core doesn't touch it */
        struct dev_pm_info      power;
        struct dev_pm_domain    *pm_domain;
#ifdef CONFIG_NUMA
        int             numa_node;      /* NUMA node this device is close to */
#endif
        u64             *dma_mask;      /* dma mask (if dma'able device) */
        u64             coherent_dma_mask;/* Like dma_mask, but for
                                             alloc_coherent mappings as
                                             not all hardware supports
                                             64 bit addresses for consistent
                                             allocations such descriptors. */
        struct device_dma_parameters *dma_parms;
        struct list_head        dma_pools;      /* dma pools (if dma'ble) */
        struct dma_coherent_mem *dma_mem; /* internal for coherent mem override */
        /* arch specific additions */
        struct dev_archdata     archdata;
        struct device_node      *of_node; /* associated device tree node */
        dev_t                   devt;   /* dev_t, creates the sysfs "dev" */
        spinlock_t              devres_lock;
        struct list_head        devres_head;
        struct klist_node       knode_class;
        struct class            *class;
        const struct attribute_group **groups;  /* optional groups */
        void    (*release)(struct device *dev);
};
```

Ohjelmalistaus 5. Device-tietorakenteen esitys.

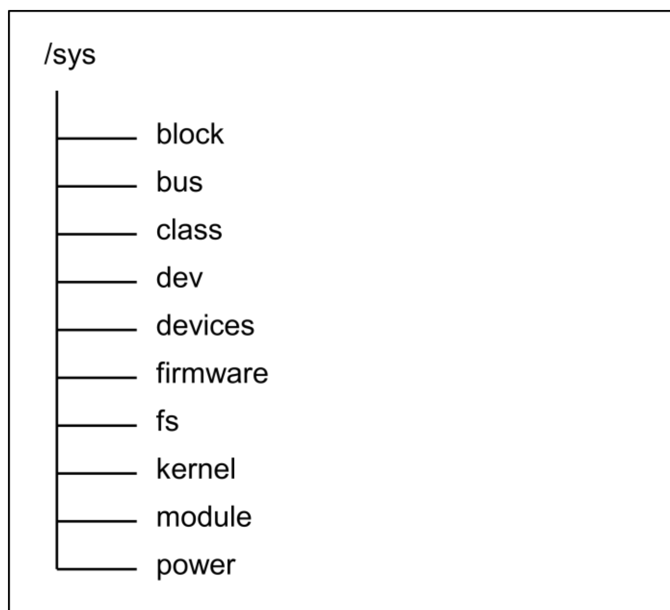
Käytännön laiteajureiden toteutuksessa harvoin joutuu operoimaan suoraan *device* tietorakenteen kanssa, vaan toteutettavat laiteajurit sidotaan liitettävään väylään koskeviin tietorakenteisiin. Esimerkkinä olkoon PCI-väylän laitetta kuvaava tietorakenne *pci_dev*, joka on määritelty Linux API linux/pci.h-tiedostossa. Tämä tietorakenne kääriI

ohjelmalistaus 6:n *device* tietorakenteen, sekä muita PCI-väylän laitetta koskevia datakenttiä. Ohjelmalistaus 6:ssa on esitettynä *pci_dev* tietorakenteen muutamat osat sekä rakenteen käyttämät ajurin rekisteröintifunktiot. Tämä *pci_dev*-tietorakenne kääritään omiin PCI-väylään liitettävien laitteiden ajuritoteutuksiin. Vastaavat rakenteet löytyvät jokaiselle järjestelmän väyläratkaisulle, jota vasten toteutetaan omat väylään liitettävät laiteajurit. [18, s. 173–175.]

```
#include <linux/pci.h>

struct pci_dev
{
    struct pci_driver *driver;
    struct device dev;
    /*...*/
};

struct pci_driver
{
    struct device_driver driver;
    /*...*/
};

int pci_register_driver(struct pci_driver *);
void pci_unregister_driver(struct pci_driver *);
```

Ohjelmalistaus 6. PCI:n käyttämät tietorakenteet ja rekisteröintifunktiot.

Unified Device Modelin yhteydessä käytetty Sysfs on virtuaalinen keskusmuistissa sijaitseva tiedostojärjestelmä. Proc-tiedostojärjestelmän tavoin Sysfs liittyy /sys hakemistoon sekä tuottaa hierarkkisen näkymän, jolla käyttäjät voivat tarkastella järjestelmään liitettyjen laitteiden topologioita yksinkertaisten tekstitiedostojen muodossa. Proc:sta poiketen Sysfs käyttää device modelin kobject-tietueita tiedostojärjestelmän esittämiseen. Kuvassa 8 on esitettynä Sysfs:n juurihakemisto, joka koostuu kymmenestä alihakemistosta.



Kuva 8. /sys juurihakemisto.

Block-hakemisto sisältää yhden alihakemiston jokaista järjestelmään rekisteröityä lohkolaitetta ja alihakemistoa kohden on N-määrä alihakemistoja, jotka listaavat lohkolaitteen osiot. Bus-hakemisto tuottaa näkymän kaikista järjestelmän väylistä, kuten PCI, i2c ja USB. Class-hakemistossa listataan luokkakohtaisen näkymän kaikista järjestelmään liitetyistä laitteista. Dev-hakemisto listaa järjestelmään luodut laitetiedostot. Tämä hakemisto jakautuu kahteen alihakemistoon, jossa eritellään lohko- ja karakterilaitteet toisistaan. Alihakemistot sisältävät symbolisia linkkejä major- ja minor-lukujen muodossa. Devices-hakemisto tuottaa näkymän laitteiden topologiasta esittämällä hierarkkisen näkymän laitestruktuureista, jotka sijaitsevat kernelissä. Firmware-hakemisto sisältää alhaisen tason järjestelmäkohtaiset alijärjestelmät, kuten ACPI, EDD ja EFI. Fs-hakemisto listaa käytössä olevat tiedostojärjestelmät. Kernel-hakemisto sisältää ytimen konfiguraatio-optiot sekä ytimen tilaa koskevaa informaatiota. Module-hakemisto listaa kaikki ytimeen liitetyt kernel-moduulit (laiteajurit). [18, s. 357.]

Sysfs-tiedostojärjestelmää käyttävät kobject-tietueet liittyvät tiedostojärjestelmään, joita kutsutaan attribuuteiksi. Attribuutit ovat tekstitiedostoja ja sisältävät tavanomaisesti yhden numeerisen tai sanallisen arvon per tiedosto. Näihin tekstitiedostoihin voi tiedoston oikeuksien puitteissa kohdistaa systeemikutsurajapinnan mukaisia luku- tai kirjoitusoperaatiota, joista Sysfs välittää tiedostojen I/O-operaatiot laiteajurin funktioihin, jotka ovat määriteltyinä tekstitiedostoa koskeville attribuuteille. [20.]

Attribuutin ja tiedosto-operaatioiden tietorakenteet ovat esitettynä ohjelmalistauksessa 7. Attribuutin tietorakenne määrittelee attribuutin nimen, attribuutin omistavan moduulin sekä attribuutin käyttäjäavaruuden luku- ja kirjoitusoikeudet. sysfs_ops-tietorakenne sisältää takaisinkutsufunktiot, joita vasten toteutetaan laiteajurin funktiot attribuutin luku- ja kirjoitustoimintoihin.

```
struct attribute
{
    const char *name; /* attribuutin nimi */
    struct module *owner; /* omistava moduuli*/
    mode_t mode; /* attibuutin oikeudet */
};

struct sysfs_ops
{
        ssize_t (*show)(struct kobject *, struct attribute *, char *);
        ssize_t (*store)(struct kobject *, struct attribute *, const char *, size_t);
};
```

Ohjelmalistaus 7. Attribuutin ja tiedosto-operaatioiden tietorakenteet.

Device-modelin mukaisten laiteajureiden toteutuksissa useimmiten *attribute-* ja *sysfs_ops*-tietorakenteita ei käytetä suoraan, vaan driver modelin käärerakenteita ja funktioita. Ohjelmalistaus 8:ssa on esitettynä Linux API device.h-tiedostossa määritelty *device_attribute*-tietorakenne sekä core.c-tiedostossa toteutetut *device_create_file-* ja *device_remove_file*-funktiot.

```
struct device_attribute
{
        struct attribute   attr;
        ssize_t (*show)(struct device *dev,
           struct device_attribute *attr, char *buf);
        ssize_t (*store)(struct device *dev,
           struct device_attribute *attr, const char *buf, size_t count);
};

int device_create_file(struct device *, const struct device_attribute *);
void device_remove_file(struct device *, const struct device_attribute *);
```

Ohjelmalistaus 8. Laite-attribuutti ja rekisteröintifunktiot.

Tiedoston luonti Sysfs-tiedostojärjestelmään tapahtuu kutsumalla *device_create_file*-funktiota, joka ottaa parametriksi laitetta kuvaavan *device*-tietorakenteen (ks. ohjelmalistaus 8) sekä *device_attribute* käärerakenteen. Attribuutin poistaminen tapahtuu vastaavasti kutsumalla *device_remove_file*-funktiota. Laiteajuri voi luoda yhden tai useampia attribuutteja Sysfs-hakemistoon.

7 IR-ohjainkortin toteutus

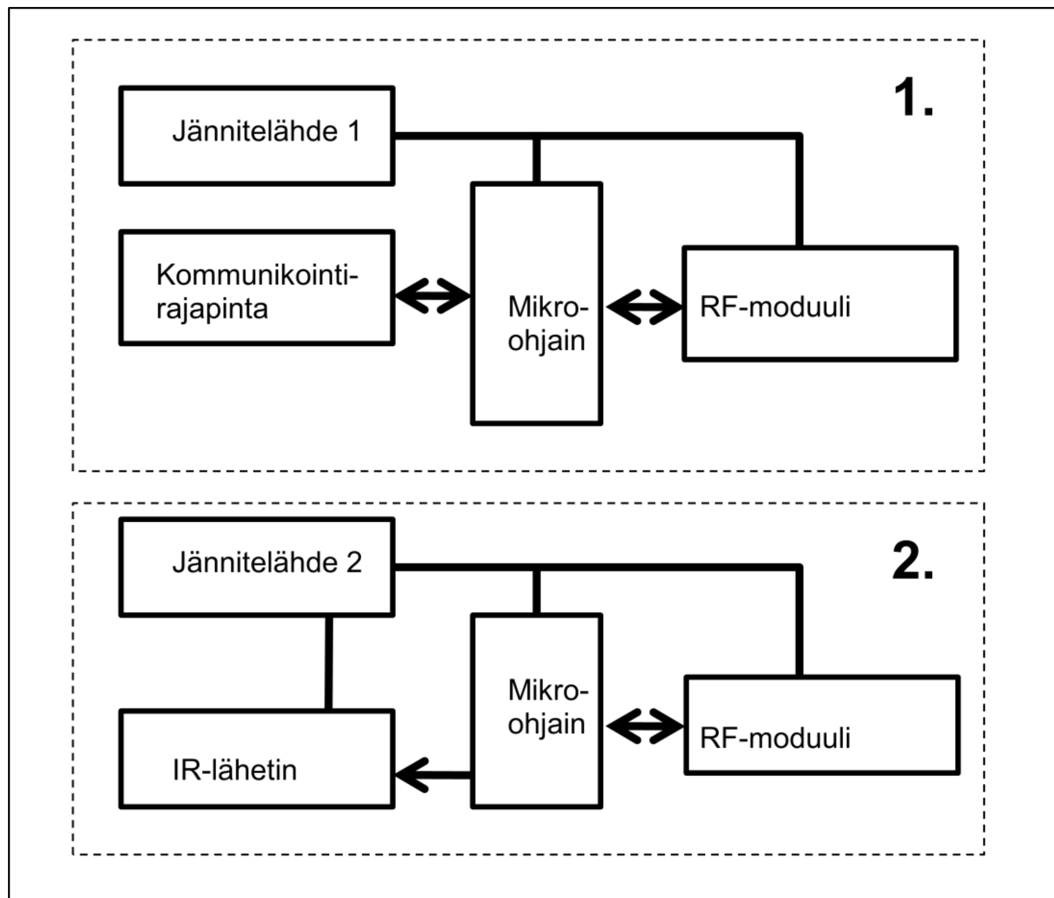
Tämä luku käsittelee IR-ohjainkortin suunnittelua ja toteutusta. Lähtökohtana on toteuttaa ohjainkortin prototyyppi, joka ensisijaisesti tulisi toimimaan Raspberry Pi -alustassa, samalla kuitenkin huomioidaan ohjainkortin siirrettävyys eri alustojen väliltä. Siirrettävyydellä tarkoitetaan, että ohjainkortti voidaan liittää ja korttia voidaan ottaa helposti käyttöön myös muissa alustoissa; esimerkiksi Windows-käyttöjärjestelmällä varustettuun pöytä -PC:hen.

Seuraavissa omissa aihekokonaisuuksissa esitetään IR-ohjainkorttia kuvaava lohkokaavio ja sen osat. Lohkokaavion jälkeen käydään läpi IR-ohjainkortin toimintakaavio, josta tulee käymään ilmi, kuinka ohjainkortin tulee toimia. Toimintakaaviota seuraa piirikaavio, jossa selvitetään ohjainkortin sähköiset ominaisuudet ja se kuinka ohjainkortin eri osat ovat fyysisesti kytköksissä toisiinsa. Luvun viimeisessä aihekokonaisuudessa selvitetään IR-ohjainkortin ohjelmiston arkkitehtuuri sekä ohjelmakoodi.

7.1 Lohkokaavio

Aloitin kortin suunnittelun lohkokaaviolla, jolla kuvataan ohjainkortin tärkeimmät osat. Ohjainkortin lohkokaavio on esitettynä kuvassa 9, joka on eritelty kahteen osaan. IR-ohjainkortti toteutettiin lohkokaavion mukaisesti, kahdesta toisistaan erillään olevasta osasta: lähettimestä ja ohjaimesta. Tämä toteutustapa nojaa loppusijoitusnäkökulmaan, jossa Raspberry Pi tai muu laite voi olla sijoitettuna paikkaan, josta järjestelmään liitetty ohjainkortti ei saa vaadittavaa suoraa näköyhteyttä ilmalämpöpumpun IR-vastaanottimeen. Ohjainkortin lähetin (kuva 9, kohta 1) tuli olemaan se osa, joka fyysisesti liitettäisiin Raspberry Pi:hin tai muuhun ohjaavaan laitteeseen ja sen tehtäväksi tulee tuli ohjauskäskyjen vastaanoton isäntälaitteelta dekoodauksen, sekä ohjauskäskyjen siirto ohjaimelle radiolinkin kautta. Ohjain (kuva 9, kohta 2) vastaanottaa lähetti-

men lähettämän viestin sekä välittää signaalin ilmalämpöpumpulle infrapunavalon muodossa. Seuraavaksi selvitetään kuvassa 9 olevan lohkokaavion osat.



Kuva 9. IR-ohjainkortin lohkokaavio.

RF-moduuli

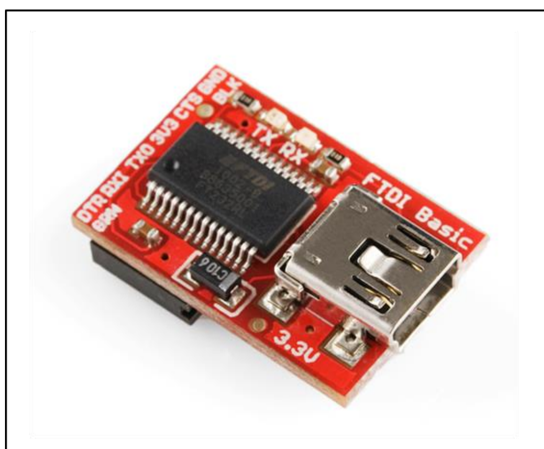
RF-moduulina käytin Texas Instrumentsin CC2500-piiriin pohjautuvaa QFM-TRX1-24G-radiomoduulia (ks. kuva 10). CC2500 tarjoaa SPI-rajapinnan, konfiguroitavat laitteen toimintaa kuvaavat ohjausrekisterit sekä datan lähetys- ja vastaanottorekisterit. CC2500 soveltuu käytettäväksi kulutuselektroniikassa, voi toimia lähettimenä, tai vastaanottimena käyttämällä jatkuvaa tai pakettimuotoista tiedonsiirtoa. Laite on määritelty pienivirtaiseksi piiriksi, joka lähetysmoodissa voi yltää 50 metrin kantamaan.

Kuva 10. QFM-TRX1-24G -radiomoduuli.

Kommunikointirajapinta

IR-ohjainkortin kommunikointirajapinnaksi valitsin sarjaportin, jonka yhteydessä käytetään universaalia asynkronista sarjaprotokollaa. Valintaa puolsi sarjaportin yksinkertaisuus sekä se, että sulautetuissa järjestelmissä laitteiden välinen keskustelu UART:n välityksellä on hyvin yleistä. Modernit työpöytäkoneet eivät tue fyysistä sarjaporttia, ja tämän rajoitteen voi kiertää esittämällä laite USB-oheislaitteena käyttämällä FTDI:n FT232RL -piiriä. FT232RL toiminnaltaan emuloi TTL-logiikkatasoista sarjaporttia toimimalla USB-oheislaitteena. Näin ollen IR-ohjainkortin helppo siirrettävyys on mahdollista kaikkien USB:tä tukevien järjestelmien väliltä.

FT232RL-piirejä on saatavilla ainoastaan pintaliitokselle tarkoitettuja QFN ja SSOP koteloissa. Tämä vaikeutti prototyypin toteutusta ja päädyin hankkimaan valmiin FTDI-Basic (ks. kuva 11) moduulin yhdysvaltalaiselta harrastelijaelektroniikkayritykseltä Sparkfun Electronics.



Kuva 11. FTDI Basic.

Mikro-ohjain

Infrapunalähettimen ohjaukseen ja kommunikointiin laiterajapinnan väliltä käytin Atmel AVR Tiny -perheeseen kuuluvaa ATtiny4313-PU mallin mikrokontrolleria. ATtiny4313 tarjoaa lähettimen kehitystyön kannalta olennaisia integroituja piirejä: kuten USART (Universal Synchronous Asynchronous Receiver Transmitter) ja USI (Universal Serial Interface). USART-piirillä voidaan toteuttaa sarjaportti, josta mikro-ohjain voi keskustella kommunikointirajapinnan kanssa asynkronisella sarjaprotokollalla. USI-piirillä voidaan toteuttaa RF-moduulin vaatima SPI-väylä.

Jännitelähde 1 ja 2

Toteutin IR-ohjainkortin jännitelähteen 1 käyttämällä FTDI Basic-moduulin USB -liitännän tarjoamaa +5 V:n jännitelähdettä. Jännitelähde 2 koostui paristoista.

IR-lähetin

IR-lähettimen tuli rakentua infrapunavaloa lähettävästä ledistä, vastuksesta sekä transistorikytkennästä, jota ohjataan mikro-ohjaimella. Käytännössä IR-lähetintä ei toteutettu, vaan toimintaa simuloitiin tavallisella näkyvää valoa lähettävällä ledillä. Näin menettelin johtuen epäselvyydestä mitä infrapunavalon aallonpituutta Mitsubishi MSZ-GE25VA:n IR-vastaanotin käytti.

7.2 IR-ohjainkortin toimintakaavio

IR-ohjainkortin lähettimen ja ohjaimen toimintakaaviot ovat esitettynä liitteissä 1 ja 2. IR-ohjainkortin lähettimen suoritus alkaa toimintakaavion mukaisesti radiomoduulin ja kommunikointirajapinnan alustustoimenpiteillä. Lähettimen alustustoimenpiteiden jälkeen, systeemi menee lepotilan odottamaan merkkijonoja kommunikointirajapinnalta. Lähetin vastaanottaa dataa tavu kerrallaan, kunnes lähetetään ASCII-merkistön "Enter"-painiketta vastaavan lukuarvon. Tämän jälkeen lähetin välittää paketin ohjaimelle.

IR-ohjainkortin ohjaimen suoritus alkaa lähettimen tavoin laitteiston alustuksella, jonka jälkeen ohjain siirtyy lepotilaan odottamaan komentoa radiomoduulilta. Paketin saapuessa välittimeltä ohjain lähettää vastaanotetun paketin IR-lähettimen lähetettäväksi.

7.3 IR-ohjainkortin piirikaavio ja sähköiset ominaisuudet

IR-ohjainkortin lähettimen ja ohjaimen piirikaaviot ovat esitettynä liitteessä 3. Tarkastellaan ensimmäiseksi lähettimen piirikaaviota. Lähettimen piirikaavio rakentuu kuvan 9 lohkokaavion kohdan 1 mukaisesti jännitelähteestä, kommunikointirajapinnasta, mikro-ohjaimesta sekä radiomoduulista. Näistä jännitelähde ja kommunikointirajapinta ovat yhdistettynä samaan FT232RLSSOP-piiriin. FT232RLSSOP-piiri toimii jännitelähteenä lähettimen osille käyttämällä USB-rajapinnan +5 V:n jännitelähdettä.

Seuraavaksi tarkastellaan liitteen 3 IR-ohjainkortin ohjaimen piirikaaviota. Ohjain rakentuu kuvan 9 lohkokaavion kohdan 2 mukaisesti mikro-ohjaimesta, IR-lähettimestä sekä radiomoduulista. Lohkokaaviosta poiketen, jännitelähteen toteutus on jätetty pois. Koska IR-ohjainkortin ohjaimen on ajateltu ottavan käyttöjännitteensä paristoista, joiden jännitealue voi vaihdella +1,8 V:n –+3,6 V:n väliltä, tämä jännitealue on ohjaimen piirien minimi- ja maksimijännite.

Lopuksi tarkastellaan IR-ohjainkortin ohjaimen IR-lähetintä. IR-lähetin rakentuu transistorin, kahden vastuksen ja IR-ledin yhteiskytkennästä. Transistori on otettu mukaan ohjaamaan lediä, koska IR-ledejä ohjataan niiden nimellistä 20–40 mA virrankestoa monta kertaa suuremmilla virtapiikeillä, tällöin voidaan parantaa signaalin kantamaa. ATTINY4313 kykenee syöttämään tai nielemään maksimissaan 40 mA pinniä kohden [22, s. 198] eikä tämä ei riitä IR-ledin ohjaamiseen. Toteuttamalla transistorikytkentä voidaan pienellä ohjausvirralla toteuttaa suuremman virran kuluttavien piirien ohjaus.

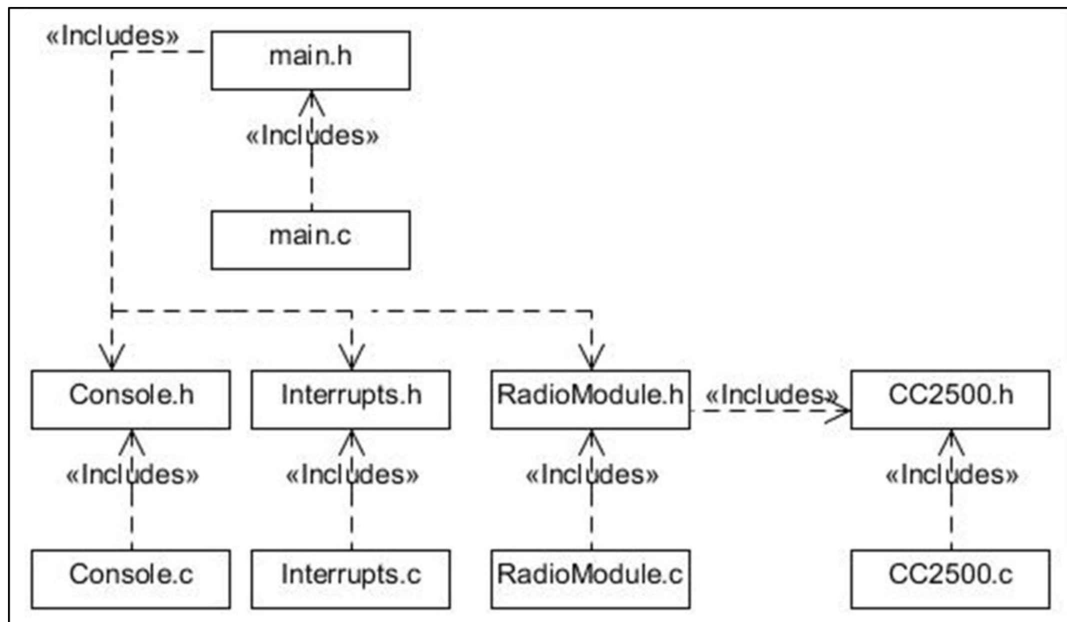
Käytännön arvot IR-lähettimen toteutukseen jäivät mittaamatta aikaisemmin mainitun epäselvyyden vuoksi, minkä tyyppisen IR-led piti valita, jotta saataisiin yhteensopivuus Mitsubishi MSZ-GE25VA IR-vastaanottimen kanssa.

7.4 Mikro-ohjainten ohjelmakoodi

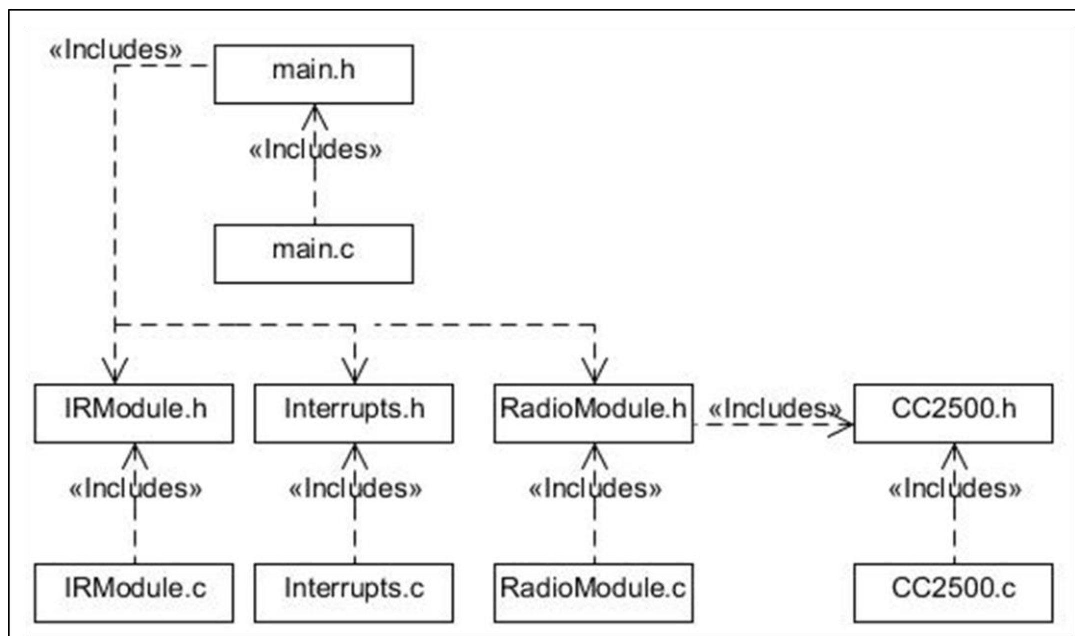
IR-ohjainkortin lähettimen ja ohjaimen C-kielellä kirjoitetut ohjelmakoodit ovat listattuna liitteissä 4–19. Kuvissa 12 ja 13 ovat esitettynä IR-ohjainkortin yksinkertaistetut lähdekoodien rakenteet. Ohjelmakoodi on jaoteltu omiin tiedostoihinsa, joista kukin tiedosto sisältää moduulia vastaavat toteutukset.

Aloitetaan tarkastelu Kuvan 12 IR-ohjainkortin lähettimen ohjelmakoodin rakenteesta. Liitteen 19 main.c-toteutuksessa on lähettimen pääohjelma ja pääohjelman käyttämät

apufunktiot. Pääohjelman toteutukseen kuuluu lähettimen alustustoimenpiteet sekä keskeytyspyyntöjen käsittely.



Kuva 12. IR-ohjainkortin lähettimen ohjelmakoodin osat.



Kuva 13. IR-ohjainkortin ohjaimen ohjelmakoodin osat.

Liitteen 15 Console.c-tiedostoa koskeva toteutus sisältää IR-ohjainkortin lähettimen kommunikointirajapinnan ohjelmallisen toteutuksen. Toteutukseen kuuluu funktioita datan vastaanottoon sekä lähetykseen IR-ohjainkortin lähetintä käyttävälle sovelluksille. Liitteiden 9 ja 17 Interrupts.c-toteutukset käsittävät mikro-ohjaimen keskeytysten

käsittelyrutiinit ja niiden käyttämät tietorakenteet. RadioModule.c, joka on toteutettu liitteessä 7 käsittää muun muassa funktiot saapuneiden datapakettien lukemisen ja lähettämisen radiolinkin kautta. RadioModule.c toteutus toimii samalla käärerajapintana liitteessä 5 olevalle CC2500.c-laiteohjausrajapinnan toteutukselle.

Kuvassa 13 esiintyvän IR-ohjainkortin ohjaimen lähdekoodin rakenne noudattaa samaa kaavaa kuin kuvan 12 lähettimen rakenne. IR-ohjainkortin ohjaimen lähdekoodissa Console.c korvautuu liitteessä 11 esiintyvällä IRModule.c-toteutuksella. IRModule.c on runkototeutus on IR-vastaanottimen datan lähetykselle.

7.5 Ohjainkortin testaus

IR-ohjainkortin testauksen tavoitteena oli todeta ohjainkortin osien toimivuus ja asettaa pohja IR-ohjainkortin jatkokehitykselle. Testausvaihe alkoi rakentamalla kahdelle koekytkentäalustalle liitteen 3 piirikaavion mukaisen kytkennän IR-ohjainkortin lähettimestä ja ohjaimesta.

Testauksen ensimmäinen vaihe oli saada IR-ohjainkortin lähettimen kommunikointirajapinta toimimaan. Testausalustaksi valittua Raspberry Pi tietokonekorttia ei ollut saatavilla ja testi siirrettiin Mac OSX-käyttöjärjestelmällä toimivalle kannettavalle tietokoneelle. Jotta IR-ohjainkortin lähettimen kommunikointirajapinta saataisiin toimimaan, vaadittiin tähän sopiva laiteajuri. FT232RL-piirin valmistaja tarjoaa laiteajureita muun muassa Windows-, Linux- ja Mac OSX -alustoille. Laiteajurin onnistuneen asennusvaiheen jälkeen kytkin IR-ohjainkortin lähettimen USB-porttiin ja tarkistin laitteen rekisteröitymisen suorittamalla konsolissa *dmesg*-komentoa, joka tulostaa kernelin viestipuskurin. Viestipuskurin päällimmäisenä viestinä löytyi seuraava tekstikenttä: *FTDIUSBSerialDriver: 0 4036001 start - ok*. Seuraavaksi tarkistin laiteajurin rekisteröinnin /dev-hakemistossa suorittamalla *ls -lt /dev*-komennon, joka listasi laitetiedostot luontijärjestyksessä. Listauksessa löytyi seuraava laitetiedosto: *tty.usbserial-A800KD97*. Kyseessä oli FT232RL-piirin laiteajurin karakterilaitetiedosto. IR-ohjainkortin lähettimen kommunikointirajapinta oli saatu kohtaan, jossa on voitu lähettää dataa vastaanottimelle.

Testauksen viimeinen vaihe oli lähettää testidataa IR-ohjainkortin ohjaimelle, jossa tavallisella ledillä pyrittiin näyttämään vastaanotetun datan bittijono. Testaus suoritettiin käyttämällä konsolin screen-ohjelmaa. Yhteyden muodostus IR-ohjainkortin lähettimelle oli seuraava: screen /dev/*tty.usbserial-A800KD97 9600*, jossa avattiin terminaaliyh-

teys FT232RL-laiteajuriin käyttäen tiedonsiirtonopeutta 9600 bps. Terminaaliyhteyden muodostuksen jälkeen lähetin testikirjaimia IR-ohjainkortin ohjaimelle ja totesin ASCII taulukon avulla, että ohjaimelle saapuneet merkit välittyivät ledille, joka mallinsi IR-lähetintä. Lopetin kortin testauksen tähän toteamalla, että IR-ohjainkortti osoitti toiminnan merkkejä ja saavutti riittävät ehdot kortin jatkokehitykselle.

7.6 IR-ohjainkortin jatkokehitys

IR-ohjainkortin kehitystyö oli lopetettu kohtaan, jossa IR-ohjainkortin lähetin ja ohjain olivat toteutettuna koekytkentäalustoille. Ohjainkortin lähettimellä voitiin lähettää testidataa ohjaimelle, josta IR-ohjainkortin ohjaimella kyettiin vastaanottamaan lähetetty data, sekä osoittamaan vastaanotettujen datapakettien eheys. Jotta ohjainkortti saataisiin ohjaamaan tämän työn kohteena olevaa ilmalämpöpumppua, täytyy ensiksi selvittää kyseisen ilmalämpöpumpun fyysisen IR-vastaanottimen ominaisuudet: kuten mille infrapunavalon aallonpituudelle IR-vastaanotin on asetettuna, sekä lähetettävän datan kantoaallon taajuus. Fyysisten ominaisuuksien lisäksi pitää olla selvillä myös ilmalämpöpumpun käyttämä IR-protokolla, sekä varsinaiset ohjauskoodit, jotta laite saataisiin tekemään halutut toimenpiteet. Näillä edellytyksillä voidaan toteuttaa liitteen 3 IR-ohjainkortin ohjaimen piirikaavion IR-ledin ohjauskytkentä sekä luvun 7.4 kuvan 13 IRModule.c lähdekoodin lopullinen toteutus.

8 Yhteenveto

Insinöörityön tavoite oli kehittää prototyyppi laitteesta, millä voitiin toteuttaa etäohjaus MSZ-GE25VA -merkkiselle ilmalämpöpumpulle. Toteutus aloitettiin esittämällä yleiskuvaus kohdejärjestelmästä. Yleiskuvauksessa kävi ilmi järjestelmän topologia, jossa oli kuvattuna järjestelmän osat. Topologian keskeisimmässä roolissa oli IR-ohjainkortti, jonka piti mallintaa ilmalämpöpumpun omaa kaukosäädintä sekä tarjota rajapinnan ulkoista ohjausta varten. IR-ohjainkortin ohjaukseen valittiin Raspberry Pi-tietokonekortti, josta jouduttiin myös valitsemaan kehitystyön kannalta sopivin malli. Seuraavaksi tarkasteltiin Raspberry Pi:n Linux-käyttöjärjestelmänä toimivaa Arch Linuxia ja läpikäytiin tämän jakelun ominaisuuksia.

Insinöörityön teoriaosuus tuki IR-ohjainkortin kehitystyötä ja Linux-laiteajureiden teoriaa. Ensimmäiseksi tutkittiin EIA/TIA 232-standardia, joka kuvasi RS-232-sarjaportin sähköiset ominaisuudet ja sen yhteydessä käytettyä universaalia asynkronista sarjaprotokollaa. Tähän pohjautuvaa ratkaisua käytettiin IR-ohjainkortin ulkoisen ohjauksen rajapintana. SPI rajapinnan teoriassa tarkasteltiin SPI protokollan rakennetta ja ajoituskaaviota. Lisäksi huomattiin, että SPI-protokollan datakehyksien konkreettinen toteutus on monissa tapauksissa laitekohtainen. Optisen tiedonsiirron teoriassa tarkasteltiin tiedonsiirtomenetelmiä optiikan avulla, sekä esimerkkipohjaisesti esitettiin Philipsin kehittämän RC-5-protokollan toimintaa. Teoriaosuuden viimeisissä osissa tutkittiin Linux-laiteajureita ja niiden rakennetta. Linux-laiteajureissa käsiteltiin, kuinka laitteet voitiin liittää osaksi Linuxia. Lisäksi tutkittiin lähdekooditasolla Linux-laiteajurin runkoa ja laiteajureiden käännösprosessia.

Insinöörityön käytännön osuudessa toteutettiin ja testattiin IR-ohjainkortin prototyyppi. Ohjainkortin kehitystyö aloitettiin toteuttamalla lohkokaavio sekä toimintakaavio. Lohkokaaviossa kuvattiin IR-ohjainkortin keskeiset osat, joiden toiminnallinen kuvaus esitettiin toimintakaaviossa. Lohkokaavion pohjalta suunniteltiin ohjainkortin piirikaavio, jossa esitettiin komponenttitason kytkennät. Lohkokaavion, toimintakaavion ja piirikaavion yhteisvaikutuksen pohjalta toteutettiin IR-ohjainkortin ohjelmakoodiosuus. IR-ohjainkortin prototyypin valmistuessa suoritettiin ohjainkortin testaus, jossa todettiin kortin toiminta ja näin asetettiin pohja IR-ohjainkortin jatkokehitykselle.

Lähteet

1 Embedded Linux Wiki. 2012. Verkkodokumentti. Wikipedia. <http://elinux.org/File:RpiFront.jpg>. Luettu 12.9.2012.

2 Raspberry Pi Hub. 2012. Verkkodokumentti. Wikipedia. <http://elinux.org/RaspberryPiBoard>. Luettu 13.9.2012.

3 Raspberry PI. 2012. Verkkodokumentti. The Raspberry Pi Foundation. <http://www.raspberrypi.org/phpBB3/>. Luettu 13.9.2012.

4 RPi Hardware. 2012. Verkkodokumentti. Wikipedia <http://elinux.org/RPi_Hardware>. Luettu 13.9.2012.

5 Arch Linux. 2012. Verkkodokumentti. Arch Linux organization. <https://wiki.archlinux.org/index.php/Arch_Linux>. Luettu 17.9.2012.

6 Arch Linux ARM. 2012. Verkkodokumentti. Arch Linux organization. <http://archlinuxarm.org/>. Luettu 17.9.2012.

7 Slackware Linux. 2012. Verkkodokumentti. Slackware Linux, Inc <http://www.slackware.com/info/>. Luettu 18.9.2012.

8 The Arch Way. 2012. Verkkodokumentti. Arch Linux organization. <https://wiki.archlinux.org/index.php/The_Arch_Way>. Luettu 18.9.2012.

9 EIA RS 232 Standard. 2012. Verkkodokumentti Adrio Communications, Ltd. <http://www.radioelectronics.com/info/telecommunications_networks/rs232/eia-rs232-c-d-standards.php>. Luettu 20.9.2012.

10 Axelson, Jan. 2007. Serial Port Complete. Madison: Lakeview Research.

11 Serial Port. 2012. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/wiki/Serial_port#Speed>. Luettu 21.9.2012.

12 MAX220–MAX249. 2010. Datalehti. Maxim Integrated <http://datasheets.maximintegrated.com/en/ds/MAX220-MAX249.pdf>. Luettu 24.9.2012.

13 Infrared. 2012. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/wiki/Near_Infrared>. Luettu 3.10.2012.

14 Philips RC5 Infrared Transmission Protocol. 2008. Verkkodokumentti. Altoum. <http://wiki.altium.com/display/ADOH/Philips+RC5+Infrared +Transmission+Protocol>. Luettu 26.9.2012.

15 Linux. 2012. Verkkodokumentti. Wikipedia. <http://fi.wikipedia.org/wiki/Linux>. Luettu 30.9.2012.

16 Corbet, Rubini, & Kroah-Hartman. 2005. Linux Device Drivers 3rd edition. O'Reilly Media, Inc.

17 Love, Robert. 2010. Linux Kernel Development 3rd edition. Addison-Wesley.

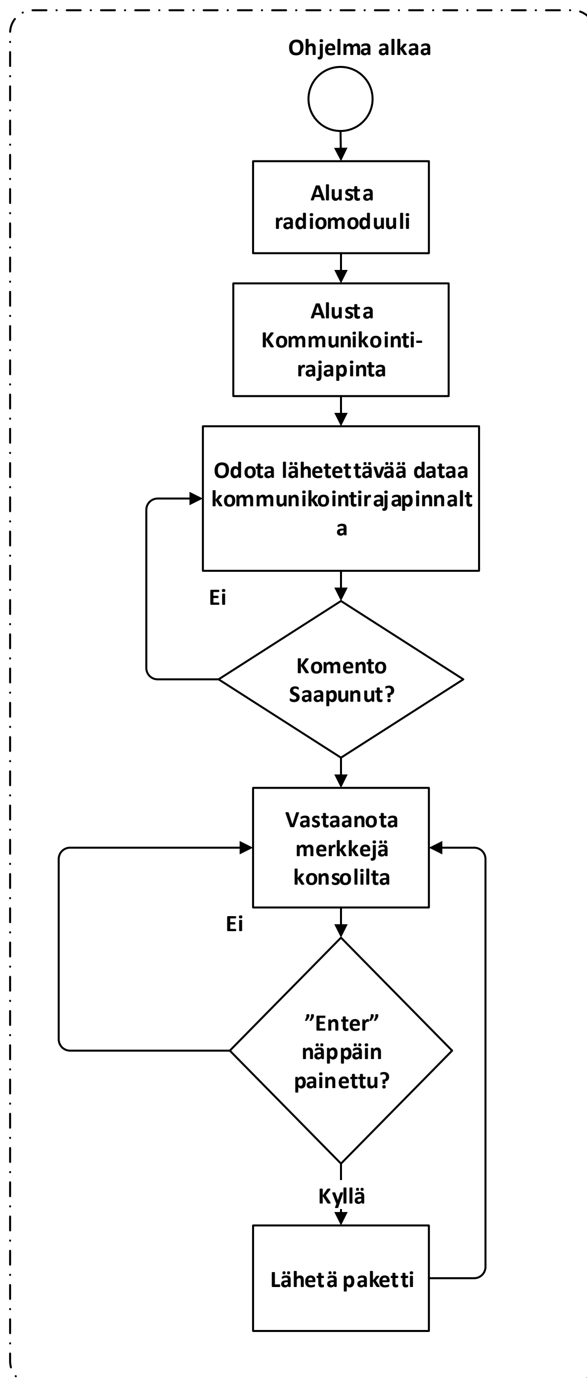
18 Cooperstein, Jerry. 2009. Writing Linux Device Drivers. Jerry Cooperstein

19 Salzman, Burian & Pomerantz. 2001. The Linux Kernel Module Programming Guide. Peter Jay Salzman

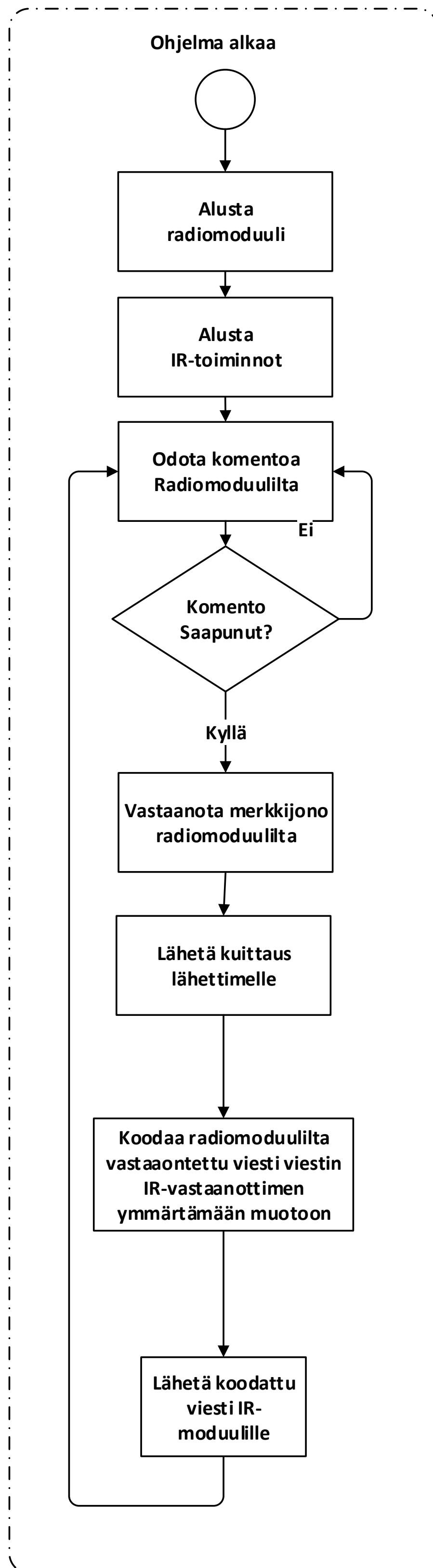
20 Sysfs. 2003. Verkkodokumentti. The Linux Kernel Organization. <https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt> Luettu 5.10.2012.

21 ATtiny2313A/4313. 2009. Datalehti. Atmel Corporate. <http://www.atmel.fi/Images/doc8246.pdf> Luettu 3.9.2013.

IR-ohjainkortin lähettimen toimintakaavio



IR-ohjainkortin ohjaimen toimintakaavio



IR-ohjainkortin piirikaavio

TITLE: IR-ohjainkortti_välittäjä | SFE | SFE

Document Number: | REV:

Date: 21.8.2013 13:55:29 | Sheet: 1/1

TITLE: IR-ohjainkortti_ohjain | SFE | SFE

Document Number: | REV:

Date: 21.8.2013 13:47:43 | Sheet: 1/1

IR-ohjainkortin CC2500.h-tiedoston lähdekoodi

```
#ifndef CC2500_H_
#define CC2500_H_

/******************THIS BLOCK DEFINE HOW DEVICE SHOULD OPERATE***************************/
/* MCU cpu settings */
#ifndef F_CPU
#define F_CPU 1000000UL
#endif

/* CC2500 SPI CS */
#define CC2500_CS_PIN     PORTB4
#define CC2500_CS_DIR     DDRB
#define CC2500_CS_PORT        PORTB

/* CC2500 register content. */
#define IOCFG2    0x29
#define IOCFG1    0x2E
#define IOCFG0    0x07
#define FIFOTHR   0x07
#define SYNC1     0xD3
#define SYNC0     0x91
#define PKTLEN                    0xFF
#define PKTCTRL1  0x04
#define PKTCTRL0  0x05
#define ADDR      0x01
#define CHANNR                    0x00
#define FSCTRL1                   0x07
#define FSCTRL0                   0x00
#define FREQ2     0x5D
#define FREQ1     0x93
#define FREQ0     0xB1
#define MDMCFG4                   0x2D
#define MDMCFG3   0x3B
#define MDMCFG2   0x73
#define MDMCFG1   0x22
#define MDMCFG0   0xF8
#define DEVIATN   0x00
#define MCSM2     0x07
#define MCSM1     0x3F
#define MCSM0     0x18
```

```
#define FOCCFG     0x1D
#define BSCFG      0x1C
#define AGCCTRL2   0xC7
#define AGCCTRL1   0x00
#define AGCCTRL0   0xB2
#define WOREVT1    0x87
#define WOREVT0                    0x6B
#define WORCTRL                    0xF8
#define FREND1                     0xB6
#define FREND0                     0x10
#define FSCAL3                     0xEA
#define FSCAL2                     0x0A
#define FSCAL1                     0x00
#define FSCAL0                     0x11
#define RCCTRL1                    0x41
#define RCCTRL0                    0x00
#define PWR_SELECT 0xFF

/* SPI operation callback*/
typedef uint8_t (*spi_readwrite_cb)(uint8_t data);

/* SPI rx pin sniff callback. */
typedef uint8_t (*spi_sniff_rx_pin_cb)(void);
/*****************************************************************************************/



/*--------CC2500 register definitions--------*/
#define CC2500_IOCFG2        0x00 // GDO2 output pin configuration
#define CC2500_IOCFG1        0x01 // GDO1 output pin configuration
#define CC2500_IOCFG0        0x02 // GDO0 output pin configuration
#define CC2500_FIFOTHR        0x03 // RX FIFO and TX FIFO thresholds
#define CC2500_SYNC1         0x04 // Sync word, high byte
#define CC2500_SYNC0         0x05 // Sync word, low byte
#define CC2500_PKTLEN                          0x06 // Packet length
#define CC2500_PKTCTRL1                        0x07 // Packet automation control
#define CC2500_PKTCTRL0                        0x08 // Packet automation control
#define CC2500_ADDR                                            0x09 // Device address
#define CC2500_CHANNR                          0x0A // Channel number
#define CC2500_FSCTRL1                         0x0B // Frequency synthesizer control
#define CC2500_FSCTRL0                         0x0C // Frequency synthesizer control
#define CC2500_FREQ2                           0x0D // Frequency control word, high byte
#define CC2500_FREQ1                           0x0E // Frequency control word, middle byte
#define CC2500_FREQ0                           0x0F // Frequency control word, low byte
```

```
#define CC2500_MDMCFG4     0x10 // Modem configuration
#define CC2500_MDMCFG3     0x11 // Modem configuration
#define CC2500_MDMCFG2     0x12 // Modem configuration
#define CC2500_MDMCFG1     0x13 // Modem configuration
#define CC2500_MDMCFG0     0x14 // Modem configuration
#define CC2500_DEVIATN     0x15 // Modem deviation setting
#define CC2500_MCSM2       0x16 // Main Radio Control State Machine configuration
#define CC2500_MCSM1       0x17 // Main Radio Control State Machine configuration
#define CC2500_MCSM0       0x18 // Main Radio Control State Machine configuration
#define CC2500_FOCCFG      0x19 // Frequency Offset Compensation configuration
#define CC2500_BSCFG       0x1A // Bit Synchronization configuration
#define CC2500_AGCCTRL2    0x1B // AGC control
#define CC2500_AGCCTRL1    0x1C // AGC control
#define CC2500_AGCCTRL0    0x1D // AGC control
#define CC2500_WOREVT1     0x1E // High byte Event 0 timeout
#define CC2500_WOREVT0     0x1F // Low byte Event 0 timeout
#define CC2500_WORCTRL     0x20 // Wake On Radio control
#define CC2500_FREND1      0x21 // Front end RX configuration
#define CC2500_FREND0      0x22 // Front end TX configuration
#define CC2500_FSCAL3      0x23 // Frequency synthesizer calibration
#define CC2500_FSCAL2      0x24 // Frequency synthesizer calibration
#define CC2500_FSCAL1      0x25 // Frequency synthesizer calibration
#define CC2500_FSCAL0      0x26 // Frequency synthesizer calibration
#define CC2500_RCCTRL1     0x27 // RC oscillator configuration
#define CC2500_RCCTRL0     0x28 // RC oscillator configuration
#define CC2500_FSTEST      0x29 // Frequency synthesizer calibration control
#define CC2500_PTEST       0x2A // Production test
#define CC2500_AGCTEST     0x2B // AGC test
#define CC2500_TEST2       0x2C // Various test settings
#define CC2500_TEST1       0x2D // Various test settings
#define CC2500_TEST0       0x2E // Various test settings


/*--------CC2500 Status register definitions--------*/
#define CC2500_PARTNUM     0x30 // CC2500 part number
#define CC2500_VERSION     0x31 // Current version number
#define CC2500_FREQEST     0x32 // Frequency offset estimate
#define CC2500_LQI         0x33 // Demodulator estimate for Link Quality
#define CC2500_RSSI        0x34 // Received signal strength indication
#define CC2500_MARCSTATE   0x35 // Control state machine state
#define CC2500_WORTIME1    0x36 // High byte of WOR timer
#define CC2500_WORTIME0    0x37 // Low byte of WOR timer
#define CC2500_PKTSTATUS   0x38 // Current GDOx status and packet status
```

```
#define CC2500_VCO_VC_DAC      0x39 // Current setting from PLL calibration module
#define CC2500_TXBYTES         0x3A // Underflow and number of bytes in the TX FIFO
#define CC2500_RXBYTES         0x3B // Overflow and number of bytes in the RX FIFO
#define CC2500_RCCTRL1_STATUS 0x3C // Last RC oscillator calibration result
#define CC2500_RCCTRL0_STATUS 0x3D // Last RC oscillator calibration result


/*--------CC2500 other access definitions--------*/
#define CC2500_PATABLE                        0x3E //patable access
#define CC2500_FIFO                                          0x3F //FIFO region access

/*--------CC2500 Command Strobes--------*/
#define CC2500_SRES          0x30 // Reset chip.
#define CC2500_SFSTXON             0x31 // Enable and calibrate frequency synthesizer (if MCSM0.FS_AUTOCAL=1).

                                 // If in RX/TX: Go to a wait state where only the synthesizer is

                                 // running (for quick RX / TX turnaround).
#define CC2500_SXOFF         0x32 // Turn off crystal oscillator.
#define CC2500_SCAL          0x33 // Calibrate frequency synthesizer and turn it off

                                 // (enables quick start).
#define CC2500_SRX           0x34 // Enable RX. Perform calibration first if coming from IDLE and

                                 // MCSM0.FS_AUTOCAL=1.
#define CC2500_STX           0x35 // In IDLE state: Enable TX. Perform calibration first if

                                 // MCSM0.FS_AUTOCAL=1. If in RX state and CCA is enabled:

                                 // Only go to TX if channel is clear.
#define CC2500_SIDLE         0x36 // Exit RX / TX, turn off frequency synthesizer and exit

                                 // Wake-On-Radio mode if applicable.
#define CC2500_SAFC          0x37 // Perform AFC adjustment of the frequency synthesizer
#define CC2500_SWOR          0x38 // Start automatic RX polling sequence (Wake-on-Radio)
#define CC2500_SPWD          0x39 // Enter power down mode when CSn goes high.
#define CC2500_SFRX          0x3A // Flush the RX FIFO buffer.
#define CC2500_SFTX          0x3B // Flush the TX FIFO buffer.
#define CC2500_SWORRST       0x3C // Reset real time clock.
#define CC2500_SNOP          0x3D // No operation. May be used to pad strobe commands to two

                                 // bytes for simpler software.
```

```
/*--------CC2500 header transmit/receive mask settings--------*/
#define CC2500_READ   0x80 //read mode
#define CC2500_WRITE  0x00 //write mode
#define CC2500_BURST  0x40 //burst mode


/*--------CC2500 chip status header info--------*/

/* Idle state
(Also reported for some transitional states instead of SETTLING or CALIBRATE)*/
#define STATUS_IDLE(H)               ((H & 0x70) == 0x00)

/* The number of bytes available in the RX FIFO or free bytes in the TX FIFO
 (See data sheet, page 44)*/
#define STATUS_FIFO_BYTES_AVAILABLE(H) ((H & 0x08)

 /* Receive mode */
#define STATUS_RX(H)                                                            ((H & 0x70) == 0x10)

 /* Transmit mode */
#define STATUS_TX(H)                 ((H & 0x70) == 0x20)

/* Frequency synthesizer is on, ready to start transmitting */
#define STATUS_FSTXON(H)             ((H & 0x70) == 0x30)

/* Frequency synthesizer calibration is running */
#define STATUS_CALIBRATE(H)          ((H & 0x70) == 0x40)

/* PLL is settling */
#define STATUS_SETTLING(H)           ((H & 0x70) == 0x50)

/* RX FIFO has overflowed. Read out any useful data, then flush the FIFO with SFRX */
#define STATUS_RXFIFO_OVERFLOW(H)      ((H & 0x70) == 0x60)

/* TX FIFO has underflowed. Acknowledge with SFTX */
#define STATUS_TXFIFO_OVERFLOW(H)      ((H & 0x70) == 0x70)

/* Stays high until power and crystal have stabilized. Should always be low when using
the SPI interface*/
#define STATUS_CHIP_RDY(H)           ((H & 0x80) == 0x80)
```

```
/*--------CC2500 function declarations--------*/
void CC2500_init(spi_readwrite_cb, spi_sniff_rx_pin_cb); //initialize CC2500 chip
void CC2500_write_register(uint8_t addr, uint8_t data); //write single register
void CC2500_write_burst(uint8_t start_addr, uint8_t *buffer, uint8_t bytes); //write multiple registers
void CC2500_read_burst(uint8_t start_addr, uint8_t *buffer, uint8_t bytes); //read multiple registers
void CC2500_sendRF_payload(uint8_t *buffer, uint8_t bytes);

uint8_t CC2500_write_strobe(uint8_t strobe); //write strobe command
uint8_t CC2500_read_register(uint8_t addr); //read single register
uint8_t CC2500_read_status_register(uint8_t addr); //read status register

#endif /* CC2500_H_ */
```

IR-ohjainkortin CC2500.c-tiedoston lähdekoodi

```
#include <avr/io.h>
#include "cc2500.h"
#include <util/delay.h>
#include <avr/pgmspace.h>

/* CC2500 private function declarations*/
static void Set_Chip_Select(uint8_t); //SPI chip select logic level
static void Wait_Rx_Pin_Low(void);
static void CC2500_burst_access(uint8_t addrAND_mode, uint8_t *buffer, uint8_t bytes);
static inline void Chip_Reset(void); // CC2500 power on reset procedure
static inline void Init_Register_Settings(void); //initialize cc2500 operation registers

static uint8_t CC2500_single_access(uint8_t addrANDmode, uint8_t data);

/*CC2500 global variables*/
spi_readwrite_cb spi_putANDread; //callback to SPI r+w implementation
spi_sniff_rx_pin_cb spi_rx_sniff; //callback cc2500 ready notify implementation

/* configuration data taken from cc2500.h */
const char configurationData[] PROGMEM =
{
        IOCFG2,  IOCFG1,  IOCFG0,   FIFOTHR,   SYNC1,
        SYNC0,   PKTLEN,  PKTCTRL1, PKTCTRL0,  ADDR,
        CHANNR,  FSCTRL1, FSCTRL0,  FREQ2,     FREQ1,
        FREQ0,   MDMCFG4, MDMCFG3,  MDMCFG2,   MDMCFG1,
        MDMCFG0, DEVIATN, MCSM2,    MCSM1,     MCSM0,
        FOCCFG,  BSCFG,   AGCCTRL2, AGCCTRL1,  AGCCTRL0,
        WOREVT1, WOREVT0, WORCTRL,  FREND1,    FREND0,
        FSCAL3,  FSCAL2,  FSCAL1,   FSCAL0,          RCCTRL1,
        RCCTRL0,
};



void CC2500_init(spi_readwrite_cb spi_rw, spi_sniff_rx_pin_cb spi_sniff)
{
        spi_putANDread = spi_rw; //store spi procedure callback
        spi_rx_sniff = spi_sniff; //store spi sniff procedure callback

        CC2500_CS_DIR |= (1 << CC2500_CS_PIN); //set CS as output IO
        Set_Chip_Select(1);//pull cs high
```

```
        Chip_Reset();//CC2500 power on reset
        Init_Register_Settings();//write cc2500 register values
}


uint8_t CC2500_write_strobe(uint8_t strobe)
{
        uint8_t status_frame;

        Set_Chip_Select(0); //cs low

        /* wait until spi rx pin goes low */
        Wait_Rx_Pin_Low();

        status_frame = spi_putANDread(strobe); //write strobe

        Set_Chip_Select(1);//cs high

        return status_frame;
}


void CC2500_write_register(uint8_t addr, uint8_t data)
{
        CC2500_single_access(addr | CC2500_WRITE, data);
}


uint8_t CC2500_read_register(uint8_t addr)
{
        return CC2500_single_access(addr | CC2500_READ, 0);
}


uint8_t CC2500_read_status_register(uint8_t addr)
{
        return CC2500_single_access(addr | CC2500_BURST | CC2500_READ, 0);
}
```

```
void CC2500_write_burst(uint8_t start_addr, uint8_t *buffer, uint8_t bytes)
{
                        CC2500_burst_access(start_addr | CC2500_WRITE | CC2500_BURST,
                                                                                                  buffer,
bytes);
}



void CC2500_read_burst(uint8_t start_addr, uint8_t *buffer, uint8_t bytes)
{
                       CC2500_burst_access(start_addr | CC2500_READ | CC2500_BURST,
                                                                                                 buffer,
bytes);
}



void CC2500_sendRF_payload(uint8_t *buffer, uint8_t bytes)
{
                       CC2500_write_strobe(CC2500_SIDLE); //set idle
                       CC2500_write_strobe(CC2500_SFTX); //flush tx fifo buffer
                       CC2500_write_register(CC2500_FIFO, bytes); //packet length
                       CC2500_write_burst(CC2500_FIFO, buffer, bytes);  //write data to FIFO

                       CC2500_write_strobe(CC2500_STX); //send packet
}



static uint8_t CC2500_single_access(uint8_t addrANDmode, uint8_t data)
{
                       uint8_t reg_content;

                       Set_Chip_Select(0); //cs low

                       /* wait until spi rx pin goes low */
                       Wait_Rx_Pin_Low();

                       spi_putANDread(addrANDmode); //write addr
```

```
        reg_content = spi_putANDread(data); //read data

        Set_Chip_Select(1);//cs high

        return reg_content;
}



static void CC2500_burst_access(uint8_t addrAND_mode, uint8_t *buffer, uint8_t bytes)
{
        uint8_t i = 0; //buffer index

        Set_Chip_Select(0); //cs low

        /* wait until spi rx pin goes low */
        Wait_Rx_Pin_Low();

        spi_putANDread(addrAND_mode); //write address and mode

        if(addrAND_mode & CC2500_READ) //if read mode
        {
                while(i < bytes)
                {
                        *(buffer + i) = spi_putANDread(0x00);
                        i++;
                }
        }
        else //write mode
        {
                while(i < bytes)
                {
                        spi_putANDread(*(buffer + i));
                        i++;
                }
        }

        Set_Chip_Select(1);
}



static void Set_Chip_Select(uint8_t pin_value)
```

```
{
		_delay_us(200);

		if(pin_value)
		{
				CC2500_CS_PORT |= (1 << CC2500_CS_PIN); //pull CS high
		}
		else
		{
				CC2500_CS_PORT &= ~(1 << CC2500_CS_PIN); //pull CS low
		}

		_delay_us(200);
}


static void Wait_Rx_Pin_Low(void)
{
		/* wait until spi rx pin goes low */
		while(spi_rx_sniff())
		{
		}
}



/* CC2500 reset procedure.
  See CC2500 manual, page 40 */
static inline void Chip_Reset(void)
{
		/* cc2500 reset chip select toggle */
		CC2500_CS_PORT &= ~(1 << CC2500_CS_PIN); //pull CS low
		_delay_us(2);
		CC2500_CS_PORT |= (1 << CC2500_CS_PIN); //pull CS high
		_delay_us(40);

		CC2500_write_strobe(CC2500_SRES); //reset chip
		CC2500_write_strobe(CC2500_SIDLE); //set chip in idle state
}



static inline void Init_Register_Settings(void)
```

```
{
        uint8_t i = 0, j = sizeof(configurationData);

        Set_Chip_Select(0); //cs low

        //wait until spi rx pin goes low
        Wait_Rx_Pin_Low();

        spi_putANDread(CC2500_IOCFG2 | CC2500_WRITE | CC2500_BURST); //write address
in burst mode

        //write register contents
        while(i < j)
        {
                spi_putANDread(pgm_read_byte((configurationData + i)));
                i++;
        }

        Set_Chip_Select(1);

        CC2500_write_register(CC2500_PATABLE, PWR_SELECT);
}
```

IR-ohjainkortin RadioModule.h-tiedoston lähdekoodi

```
#ifndef RADIOMODULE_H_
#define RADIOMODULE_H_

#define RADIOMODULE_NOERR 0
#define RADIOMODULE_ERR -1

void RadioModule_Initialize(void);
void RadioModule_SetRXMode(void);
void RadioModule_SetTXMode(void);
void RadioModule_PacketReceivedNotificationEnable(uint8_t enable);
uint8_t RadioModule_TestConnection(void);
uint8_t RadioModule_WritePayload(uint8_t *char_arr, uint8_t size);
uint8_t RadioModule_ReadRXData(uint8_t *data_array, uint8_t arraySize);

#endif /* RADIOMODULE_H_ */
```

IR-ohjainkortin RadioModule.c-tiedoston lähdekoodi

```
#define F_CPU 1000000UL

#include <avr/io.h>
#include <util/delay.h>
#include "RadioModule.h"
#include "cc2500.h"

//RadioModule GDO 2 pin definitions
#define CC2500_GDO2_DIR  DDRD
#define CC2500_GDO2_PORT PORTD
#define CC2500_GDO2_PIN  PORTD4

//RadioModule GDO 0 pin definitions
#define CC2500_GDO0_DIR  DDRD
#define CC2500_GDO0_PORT PORTD
#define CC2500_GDO0_PIN  PORTD2

//SPI pin definitions
#define CLK  PORTB7
#define DO   PORTB6
#define DI   PORTB5

void Initialize_SPI(void);
void Initialize_RadioModule_GPIO(void);
void SPI_Pulse_Delay(void);
uint8_t SPI_Write_And_Read(uint8_t data);
uint8_t CC2500_GDO2_Pin_Test(void);


/**
 * \brief Initialize Radio module.
 *
 * \param
 *
 * \return void
 */
inline void RadioModule_Initialize(void)
{
		Initialize_RadioModule_GPIO();
		Initialize_SPI();
		CC2500_init(&SPI_Write_And_Read, &CC2500_GDO2_Pin_Test);
```

```
}


/**
 * \brief SPI module initialization routine
 *
 * \param
 *
 * \return void
 */
inline void Initialize_SPI(void)
{
        DDRB  |=  (1 << CLK) | (1 << DO);        //CLK and MOSI as output io
        DDRB  &= ~(1 << DI);
//MISO as input io
        PORTB &= ~((1 << CLK) | (1 << DO) | (1 << DI)); // set output as 0

        USICR = (1 << USIWM0); //3-wire mode
}


inline void Initialize_RadioModule_GPIO(void)
{
        /* Radio module device ready pin config */
        CC2500_GDO2_DIR  &= ~(1 << CC2500_GDO2_PIN); //pin PD4 pin as input
        CC2500_GDO2_PORT &= ~(1 << CC2500_GDO2_PIN); //disable pull-up resistor

        /* Radio module data packed received notification pin config */
        CC2500_GDO0_DIR &= ~(1 << CC2500_GDO0_PIN); //pin PD2 as input
        CC2500_GDO0_PORT &= ~(1 << CC2500_GDO0_PIN); //disable pull-up resistor

        MCUCR |= (1 << ISC01) | (1 << ISC00); // pin PD2 rising edge generate interrupt(INT0)
        GIMSK |= (1 << INT0); //enable INT0 interrupt generation this is meant for CC2500 packet
received interrupt generation

}


/**
 * \brief Enable interrupt generation when cc2500 receives packet
 *
 * \param enable 1 enable 0 disable
 *
```

```
 * \return void
 */
void RadioModule_PacketReceivedNotificationEnable(uint8_t enable)
{
	if(enable)
	{
		GIMSK |= (1 << INT0); //enable INT0 interrupt generation this is meant for
CC2500 packet received interrupt generation
	}
	else
	{
		GIMSK &= ~(1 << INT0); //disable INT0 interrupt generation this is meant
for CC2500 packet received interrupt generation
	}
}


/**
 * \brief Set radiomodule into receive mode
 *
 * \param
 *
 * \return void
 */
void RadioModule_SetRXMode(void)
{
	CC2500_write_strobe(CC2500_SFRX); //flush
	CC2500_write_strobe(CC2500_SRX); //set rx mode
}


/**
 * \brief Set radiomodule into transmit mode
 *
 * \param
 *
 * \return void
 */
void RadioModule_SetTXMode(void)
{
	CC2500_write_strobe(CC2500_SFTX); //flush
	CC2500_write_strobe(CC2500_STX); //set tx mode
}
```

```
/**
 * \brief Delay generated to delay SPI signal pulse width. Function will software wait 1ms
 *
 * \param
 *
 * \return void
 */
void SPI_Pulse_Delay(void)
{
		_delay_ms(1);
}



/**
 * \brief Test connection with listening RF module. Not Implemented.
 *
 * \param
 *
 * \return uint8_t 0 if ok -1 if no reply
 */
uint8_t RadioModule_TestConnection(void)
{
		//TODO: Implementation
		return RADIOMODULE_NOERR;
}


/**
 * \brief Send RF message to listening RF module.
 *
 * \param char_arr array to send via Radio module
 * \param size actual elements to send in array, for example array size 5, but hold only 3 values that need
to be sent.
 *
 * \return uint8_t 0 if -1 if error happened
 */
uint8_t RadioModule_WritePayload(uint8_t *char_arr, uint8_t size)
{
		/* Test connection before sending data */
		if(RadioModule_TestConnection() == RADIOMODULE_NOERR)
```

```
		{
			CC2500_sendRF_payload(char_arr, size);
		}

		//TODO: Implement acknowledgment wait from receiver

		return RADIOMODULE_NOERR;
}


/**
 * \brief Read received data from device buffer
 *
 * \param data_array Where to store data
 * \param arraySize size of provided array
 *
 * \return uint8_t amount of bytes read+
 */
uint8_t RadioModule_ReadRXData(uint8_t *data_array, uint8_t arraySize)
{
		uint8_t   bytesRead  =  CC2500_read_status_register(CC2500_RXBYTES);  //amount  of
bytes in rx register
		bytesRead = bytesRead & 0x7F; //only lower 7 bits are valid

		if(bytesRead > arraySize) //do not read more than size of array
		{
			bytesRead = arraySize;
		}
		CC2500_read_burst(CC2500_FIFO, data_array, bytesRead);
		return bytesRead;
}


/**
 * \brief SPI implementation method to CC2500 interface
 *
 * \param data byte to send via SPI interface
 *
 * \return uint8_t data received during write.
 */
uint8_t SPI_Write_And_Read(uint8_t data)
{
		USIDR = data;
```

```
        USISR = (1 << USICNT3) | (1 << USIOIF);

        while(!(USISR & (1 << USIOIF)))
        {
                SPI_Pulse_Delay();
                USICR |= (1 << USITC);
                SPI_Pulse_Delay();
                USICR |= (1 << USITC) | (1 << USICLK);
        }

        return USIDR;
}


/**
 * \brief Pin test implementation method for CC2500 interface. Function will sniff cc2500 pin if
 * it´s ready to receive spi messages.
 *
 * \param
 *
 * \return uint8_t Pin logical status
 */
uint8_t CC2500_GDO2_Pin_Test(void)
{
        return CC2500_GDO2_PORT & (1 << CC2500_GDO2_PIN);
}
```

IR-ohjainkortin ohjaimen Interrupts.h-tiedoston lähdekoodi

```
#ifndef INTERRUPTS_H_
#define INTERRUPTS_H_

struct irqFlags
{
		volatile uint8_t b_int0_transition : 1;
};


#endif /* INTERRUPTS_H_ */
```

IR-ohjainkortin ohjaimen Interrupts.c-tiedoston lähdekoodi

```
#include <avr/interrupt.h>
#include "Interrupts.h"

struct irqFlags irqflags =
{
		.b_int0_transition = 0,
};


/* radio module packet ready notification irq*/
ISR(INT0_vect)
{
		irqflags.b_int0_transition = 1;
}
```

IR-ohjainkortin ohjaimen IRModule.h-tiedoston lähdekoodi

```
#ifndef IRMODULE_H_
#define IRMODULE_H_

#include <avr/io.h>

void IRModule_Initialize(void);
void IRModule_SendIRData(uint8_t *dataToSend, uint8_t arrSize);

#endif /* IR-MODULE_H_ */
```

IR-ohjainkortin ohjaimen IRModule.c-tiedoston lähdekoodi

```
#ifndef F_CPU
#define F_CPU 1000000UL
#endif

#include "IRModule.h"
#include <util/delay.h>

static void IRModule_SendByteToIR(uint8_t byteTosend);

/**
 * \brief IRModule initialization
 *
 * \param
 *
 * \return void
 */
inline void IRModule_Initialize(void)
{
	PORTB &= ~(1 << PORTB2); //Pin output 0
	DDRB  |= (1 << PORTB2); //Pin direction as output
}

/**
 * \brief Send byte data via IR interface. Send msb first. Function is Unfinished
 *
 * \param dataToSend byte to send
 *
 * \return void
 */
void IRModule_SendIRData(uint8_t *dataToSend, uint8_t arrSize)
{
	uint8_t amountOfBytesToSend = dataToSend[0]; //amount of "actual" data(exclude all
status and signal variables)
	uint8_t indexer = 0;

	while(indexer < amountOfBytesToSend)
	{
		IRModule_SendByteToIR(dataToSend[indexer + 1]);
		indexer++;
	}
}
```

```
static void IRModule_SendByteToIR(uint8_t byteTosend)
{
	uint8_t indexer = 0;

	while(indexer < 8) //iterate for 0 to 7 to send every bit to IR output
	{
		if(byteTosend & 0x80) //mask MSB
		{
			PORTB |= (1 << PORTB2); //set output to 1
		}
		else
		{
			PORTB &= ~(1 << PORTB2); //set output to 0
		}

		indexer++;
		byteTosend <<= 1; //shift bits to left
		_delay_ms(500); //slow down to see actual led blink
	}

	PORTB &= ~(1 << PORTB2); //set output to 0 if stayed on 1
}
```

IR-ohjainkortin ohjaimen main.h-tiedoston lähdekoodi

```
#ifndef MAIN_H_
#define MAIN_H_

#include <avr/io.h>
#include "IRModule.h"
#include "RadioModule.h"
#include "interrupts.h"

#endif /* MAIN_H_ */
```

IR-ohjainkortin ohjaimen main.c-tiedoston lähdekoodi

```
#include "main.h"
#define RX_BUFFER_SIZE 16

void Pull_Up_Unused_GPIO(void);
void Initialize_MCU_Internals(void);
void PushRadiodataToIR(void);

uint8_t rx_buffer[RX_BUFFER_SIZE];

int main(void)
{
		extern struct irqFlags irqflags;

		Pull_Up_Unused_GPIO();//pull-up all unused MCU pins to +VCC to prevent floating
		Initialize_MCU_Internals(); //set cpu internals
		IRModule_Initialize(); //startup code for infrared transmitter
		RadioModule_Initialize();//startup code for cc2500 radio module
		RadioModule_SetRXMode();//set radiomodule to receive mode

		asm("sleep"); //wait for interrupt

   while(1)
   {
				if(irqflags.b_int0_transition) //cc2500 packet received notification caused
interrupt
				{
						PushRadiodataToIR();
						irqflags.b_int0_transition = 0; //cc2500 data ready request
serviced
				}

				asm("sleep"); //wait for next interrupt
   }
}


/**
 * \brief Set unused pins to input and enable internal pull-up resistor
 *
 * \param
 *
```

```
 * \return void
 */
inline void Pull_Up_Unused_GPIO(void)
{
		PORTA |= (1 << PORTA1) | (1 << PORTA0);
		PORTB |= (1 << PORTB3) | (1 << PORTB1) | (1 << PORTB0);
		PORTD |= (1 << PORTD6) | (1 << PORTD5) | (1 << PORTD3);
}


/**
 * \brief Set CPU low power mode, enable interrupts etc*
 *
 * \param
 *
 * \return void
 */
inline void Initialize_MCU_Internals(void)
{
		ACSR  = (1 << ACD);//disable analog comparator
		MCUCR = (0 << SM1) | (0 << SM0)  | (1 << SE); //enable idle sleep mode
		//TODO: PUT into deeper sleep mode

		asm("sei");//enable interrupts
}

/**
 * \brief Output radio message to IR
 *
 * \param
 *
 * \return void
 */
void PushRadiodataToIR(void)
{
		RadioModule_ReadRXData(rx_buffer, sizeof(rx_buffer)); //read data from buffer
		IRModule_SendIRData(rx_buffer, sizeof(rx_buffer)); // resend data as IR message

		//TODO: Next iteration of read data if not all bytes couldn't fit into buffer.
}
```

IR-ohjainkortin lähettimen Console.h-tiedoston lähdekoodi

```
#ifndef CONSOLE_H_
#define CONSOLE_H_

//ASCII definitions
#define ASCII_CR 0x0D //ASCII carriage return
#define ASCII_BS 0x08 //ASCII backspace
#define ASCII_LF 0x0A //ASCII line feed(new line)

//function forward declarations
void Console_Initialize(void);
void Console_Disable_RX_IRQ(void);
void Console_Enable_RX_IRQ(void);
void Console_Disable_TX_IRQ(void);
void Console_Enable_TX_IRQ(void);
void Console_New_Line(void);
void Console_Send_Byte(uint8_t u8_byteToSend);
void Console_Send_String(const char *MSG);

#endif /* CONSOLE_H_ */
```

IR-ohjainkortin lähettimen Console.c-tiedoston lähdekoodi

```
#include <avr/io.h>
#include <avr/pgmspace.h>
#include "Console.h"

//UART definitions
#define F_OSC 1000000UL //CPU FREQ
#define BAUD 9600 //UART comm baud rate
#define BAUD_PRESCALE ((F_OSC / (8UL * BAUD)) - 1) //9600 bps

/**
 * \brief Initialize device UART to 9600-N-8
 *
 * \param
 *
 * \return void
 */
inline void Console_Initialize(void)
{
		UBRRH = BAUD_PRESCALE >> 8;
		UBRRL = BAUD_PRESCALE;

		UCSRA = (1 << U2X);
		UCSRB = (1 << RXEN)  | (1 << TXEN);
		UCSRC = (1 << UCSZ1) | (1 << UCSZ0);
}


/**
 * \brief Disable received character interrupts From console.
 *
 * \param
 *
 * \return void
 */
inline void Console_Disable_RX_IRQ(void)
{
		UCSRB &= ~(1 << RXCIE); //disable RX interrupt
}


/**
```

```
 * \brief Enable received character interrupts From console.
 *
 * \param
 *
 * \return void
 */
inline void Console_Enable_RX_IRQ(void)
{
		//UCSRA &= ~(1 << RXC);//clear pending interrupts
		UCSRB |= (1 << RXCIE); //enable RX interrupt
}


/**
 * \brief Disable character transmission complete interrupts From console.
 *
 * \param
 *
 * \return void
 */
inline void Console_Disable_TX_IRQ(void)
{
		UCSRB &= ~(1 << TXCIE); //disable TX interrupt
}


/**
 * \brief Enable character transmission complete interrupts From console.
 *
 * \param
 *
 * \return void
 */
inline void Console_Enable_TX_IRQ(void)
{
		//UCSRA &= ~(1 << TXC);//clear pending interrupts
		UCSRB |= (1 << TXCIE); //enable RX interrupt
}


/**
 * \brief Send byte to UART endpoint
 *
```

```
 * \param u8_byteToSend
 *
 * \return void
 */
void Console_Send_Byte(uint8_t u8_byteToSend)
{
	while(!(UCSRA & (1 << UDRE))) //loop until UART data reg is free to send another byte
	{

	}
	UDR = u8_byteToSend;
}

/**
 * \brief Send String to UART endpoint
 *
 * \param MSG
 *
 * \return void
 */
void Console_Send_String(const char *MSG)
{
	uint8_t i = 0;
	uint8_t byte = 0;

	//enable byte transmission complete interrupt
	Console_Enable_TX_IRQ();

	byte = pgm_read_byte(MSG);

	while(byte != 0)
	{
		Console_Send_Byte(byte); //send message
		asm("sleep");//sleep until transmission is complete
		i++;
		byte = pgm_read_byte(MSG + i); //collect next byte
	}

	Console_New_Line(); //new line after printing string

	//disable byte transmission complete interrupt
	Console_Disable_TX_IRQ();
}
```

```
/**
 * \brief Write new line to console interface.
 *
 * \param
 *
 * \return void
 */
void Console_New_Line(void)
{
		Console_Send_Byte(ASCII_CR);
		Console_Send_Byte(ASCII_LF);
}
```

IR-ohjainkortin lähettimen Interrupts.h-tiedoston lähdekoodi

```
#ifndef INTERRUPTS_H_
#define INTERRUPTS_H_

#define UARTBUFFER_RESET 0xFF

struct IRQFlags
{
	volatile uint8_t b_charFromConsoleReceived : 1;
	volatile uint8_t b_int0_transition		: 1;
};

//represents message buffer from UART(Console)
struct UARTBuffer
{
	uint8_t u8arr_cbuffer[32];
	volatile uint8_t u8_indexer;
};

#endif /* INTERRUPTS_H_ */
```

IR-ohjainkortin lähettimen Interrupts.c-tiedoston lähdekoodi

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "Interrupts.h"
#include "Console.h"

struct IRQFlags irq_Flags =
{
		.b_charFromConsoleReceived = 0,
		.b_int0_transition = 0,
};


struct UARTBuffer s_uBuffer =
{
		.u8_indexer = UARTBUFFER_RESET, //0xFF to start preindexing from zero
};


/* Byte received from UART IRQ handler */
ISR(USART_RX_vect)
{
		uint8_t udr_data = UDR; //collect received byte and clear pending interrupt
		uint8_t index = (s_uBuffer.u8_indexer) + 1;

		if(index < sizeof(s_uBuffer.u8arr_cbuffer)) //add char to buffer if not full
		{
				s_uBuffer.u8_indexer++;
				s_uBuffer.u8arr_cbuffer[s_uBuffer.u8_indexer] = udr_data;
				irq_Flags.b_charFromConsoleReceived = 1; //inform main that this isr
was serviced and character accepted
		}
}

/* Byte transmitted from UART IRQ Handler*/
ISR(USART_TX_vect)
{
		//Do nothing just wake up cpu
}
```

IR-ohjainkortin lähettimen main.h-tiedoston lähdekoodi

```
#ifndef MAIN_H_
#define MAIN_H_

//Includes
#include <avr/io.h>
#include <avr/pgmspace.h>

#include "Console.h"
#include "RadioModule.h"
#include "Interrupts.h"

#endif /* MAIN_H_ */
```

IR-ohjainkortin lähettimen main.c-tiedoston lähdekoodi

```
#include "main.h"

//Public variables
//variable location interrupts.c
extern struct IRQFlags irq_Flags;  //flag set by corresponding ISR to help drive main program
extern struct UARTBuffer s_uBuffer; //Characters received from console via UART

//Reply strings which are sent to console interface
const char OK[] PROGMEM = "0\0";
const char ERROR[] PROGMEM = "-1\0";

//Functions forward declarations
void Pull_Up_Unused_GPIO(void);
void Initialize_MCU_Internals(void);
void Process_Command(struct UARTBuffer *buffer);
void Process_Console_Character(uint8_t u8_Character);
void Print_Error_Code(const uint8_t u8_errCode);

int main(void)
{
		Pull_Up_Unused_GPIO();//pull-up all unused MCU pins to +VCC to prevent floating
		Console_Initialize();//startup code for MCU UART
		Initialize_MCU_Internals();//set up MCU sleep modes/disable unused modules/enable IRQ
handle
		RadioModule_Initialize();//startup code for cc2500 radio module
		Console_Enable_RX_IRQ();//enable character receive IRQ from UART

		asm("sleep"); //wait for interrupt

		//interrupt driven main program
		//software execution will base on external flags set by interrupt handlers.
    while(1)
    {
				//Console caused interrupt by receiving character from UART
				if(irq_Flags.b_charFromConsoleReceived)
				{
						//disable character receive IRQ from UART during char-
acter processing
						Console_Disable_RX_IRQ();
```

```
                        irq_Flags.b_charFromConsoleReceived = 0; //clear con-
sole interrupt notification

                        //check if received character is "Enter" key press or
something else
                        if(s_uBuffer.u8arr_cbuffer[s_uBuffer.u8_indexer]      ==
ASCII_CR)
                        {
                                Process_Command(&s_uBuffer);   //start
processing command string
                                s_uBuffer.u8_indexer   =   UARTBUFF-
ER_RESET; //reset character buffer indexer
                        }
                        else //key press was not "Enter" key
                        {
                                //Echo character back to console inter-
face
                                Con-
sole_Send_Byte(s_uBuffer.u8arr_cbuffer[s_uBuffer.u8_indexer]);

                                //if received character is "backspace"

        if(s_uBuffer.u8arr_cbuffer[s_uBuffer.u8_indexer] == ASCII_BS)
                                {
                                        s_uBuffer.u8_indexer  -=
0x02;//"remove" backspace char and latest character from buffer
                                }
                        }

                        //allow character received interrupt from console
                        Console_Enable_RX_IRQ();
                }

                asm("sleep"); //wait for next interrupt
    }
}


/**
 * \brief Set unused pins to input and enable internal pull-up resistor *
 *
 * \param
 *
```

```
 * \return void
 */
inline void Pull_Up_Unused_GPIO(void)
{
			PORTA |= (1 << PORTA1) | (1 << PORTA0);
			PORTB |= (1 << PORTB3) | (1 << PORTB1) | (1 << PORTB0);
			PORTD |= (1 << PORTD6) | (1 << PORTD5) | (1 << PORTD3) | (1 << PORTD1) | (1 <<
PORTD0);
}


/**
 * \brief Set CPU low power mode, enable interrupts etc.
 *
 * \param
 *
 * \return void
 */
inline void Initialize_MCU_Internals(void)
{
			ACSR  = (1 << ACD);//disable analog comparator
			PRR   = (1 << PRTIM1) | (1 << PRTIM0); //Disable timer/counter 1 & 0
			MCUCR = (0 << SM1) | (0 << SM0)  | (1 << SE); //enable idle sleep mode

			asm("sei");//enable interrupts
}


/**
 * \brief Process command according to message received from console interface
 *
 * \param
 *
 * \return void
 */
void Process_Command(struct UARTBuffer *buffer)
{
			Console_New_Line(); //print new line
			Print_Error_Code(RadioModule_WritePayload(buffer->u8arr_cbuffer,			buffer-
>u8_indexer));
}
```

```
/**
 * \brief This function is for writing result message to console based on radio module operation result
 * \param u8_errCode constant error code.
 *
 * \return void
 */
void Print_Error_Code(const uint8_t u8_errCode)
{
		if(u8_errCode == RADIOMODULE_NOERR)
		{
				Console_Send_String(OK);
		}
		else
		{
				Console_Send_String(ERROR);
		}
}
```