Tampereen yliopisto

Alexei Trofimov

BLUETOOTH- JA BLE-TEKNOLOGIOIDEN TOTEUTUS ESP32:LLA

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Yliopistonlehtori Erja Sipilä
Joulukuu 2022

TIIVISTELMÄ

Alexei Trofimov: Bluetooth- ja BLE-teknologioiden toteutus ESP32:lla
Kandidaatintyö
Tampereen yliopisto
Tietotekniikka
Joulukuu 2022

Työn tarkoituksena on selvittää, mistä protokollista Bluetooth- ja energiatehokkaan Bluetooth-teknologian (engl. Bluetooth Low Energy, BLE) pinot koostuvat sekä mikä on näiden protokollien tehtävä. Tämän lisäksi selvitetään, miten kyseinen pino on toteutettu ESP32-pohjaisella mikrokontrollerilla ja miten sen ohjelmointi tapahtuu käyttämällä Espressif-esineiden internetin viitekehystä (engl. Espressif IoT Development Framework, ESP-IDF).

ESP32 on mikrokontrolleri, jonka tärkeimpiin toiminnallisuuksiin kuuluu Bluetooth 4.2 sekä WiFI-tuki. Näiden lisäksi ESP32:lla on myös tuki yleisimmille mikrokontrollereissa esiintyville liitäntäspesifikaatioille, joiden avulla mikrokontrolleri voidaan esimerkiksi yhdistää osaksi isompaa kokonaisuutta. ESP:n ohjelmoimiseen on useita eri työkaluja erilaisten käyttäjien tarpeeseen. Työkalut voivat erota toisistaan sen mukaan, mitkä ovat niiden kanssa käytettävät ohjelmointikielet sekä tarjolla olevat erilaisten lisälaitteiden ja komponenttien kirjastot.

Bluetooth on langattoman tiedonsiirron tekniikka, jolla aikoinaan pyrittiin korvaamaan laitteiden välinen fyysinen linkki. Bluetooth on toteutettu usean protokollan muodostamana pinona, jotka yhdessä mahdollistavat sen toiminnan. Näitä ovat fyysisen kerroksen toiminnasta vastaavat radioprotokolla ja kantataajuusprotokolla. Korkeamman tason protokollat kommunikoivat fyysisen tason kanssa isäntäohjainrajapinnan kautta. Se mahdollistaa esimerkiksi korkeamman ja fyysisen tason protokollien toiminnallisuuden erottamisen erillisiin laitteisiin. Bluetoothin loppukäyttäjälle näkyvä toiminnallisuus on toteutettu käyttämällä profiileja. Profiileilla pystytään yhtenäistämään Bluetooth-laitteiden toiminnallisuus standardoimalla se. Tämä mahdollistaa eri valmistajien laitteiden keskinäisen toiminnan ilman ylimääräisiä asetuksia.

Bluetooth-toiminnallisuuden ohjelmoiminen on tehty ESP-IDF:ssä helpoksi käyttämällä sen tarjoamia funktioita. Työssä käytetään hyväksi ESP-IDF:n käyttämiä NimBLE- ja Bluedroid-pinojärjestäjiä. Pinojärjestäjät hallinnoivat Bluetooth-pinon ohjelmistototeutusta sekä toimivat välilinkkinä profiilien ja fyysisen kerroksen ohjelmistolle. Bluetooth-rajapintaa tutkiessa huomattiin, että koko Bluetooth-pinon ohjelmistototeutus ei ole kuitenkaan saatavilla ESP-IDF:n rajapinnan käyttäjälle, sillä fyysisen kerroksen protokollien kooditoteutus annetaan pelkästään jo käännettyinä binääreinä. Näitä binäärejä voikin käyttää pelkästään pinojärjestäjien tarjoamien funktioiden kautta.

Avainsanat: Bluetooth, BLE, Bluetooth Low Energy, ESP32, mikrokontrolleri

ALKUSANAT

Kyseisen työn aihe oli valittu, kun erästä omaa projektia tehdessä heräsi suurempi mielenkiinto Bluetooth-teknologiaa kohtaan. Työtä tehdessä uskon saavuttaneeni itselle asettamani tavoitteet ja opin erittäin paljon aiheeseen liittyen.

Haluaisin kiittää etenkin kavereitani, jotka ovat tehneet koronavuosistakin kivoja ja auttaneet sekä yleisissä, että opintoihin liittyvissä asioissa. Haluaisin myös kiittää ohjaajaani Erja Sipilää hyvästä ohjaamisesta kandidaatintyön tekemisen aikana.

Tampereella, 14. joulukuuta 2022

Alexei Trofimov

SISÄLLYSLUETTELO

LYHENTEET

A2DP	Edistynyt audionvälitysprofiili (engl. Advanced Audio Distribution Profile)
ACL	Asynkroninen yhdistymis-painotteinen linkki (engl. Asynchronous Connection-Oriented Link)
ADC	Analogia-digitaalimuunnin (engl. Analog to Digital Converter)
API	Ohjelmointirajapinta (engl. Application Programming Interface)
AVRCP	Audio/video-etäohjausprofiili (engl. Audio/Video Remote Control Profile)
BLE	Energiatehokas Bluetooth-yhteys (engl. Bluetooth Low Energy)
BR	Tavallinen tiedonsiirtonopeus (engl. Basic Rate)
DAC	Digitaali-analogiamuunnin (engl. Digital to Analog Converter)
EDR	Parannettu tiedonsiirtonopeus (engl. Enhanced Data Rate)
FCS	Kehyksen tarkastusjakso (engl. Frame Check Sequence)
FSK	Vaihtotaajuusmodulaatio (engl. Frequency Shift Keying)
GAP	Yleinen pääsyprofiili (engl. Generic Access Profile)
GATT	Yleinen ominaisuusprofiili (engl. Generic Attribute Profile)
GFSK	Gauss-suodatettu vaihtotaajuusmodulaatio (engl. Gaussian Frequency-Shift Keying)
GOEP	Yleinen objektienvaihtoprofiili (engl. Generic Object Exchange Profile)
GPIO	Yleiskäyttöinen sisään- ja ulostulo (engl. General Purpose Input/Output)
HCI	Isäntäohjainrajapinta (engl. Host Controller Interface)
IDE	Integroitu ohjelmointiympäristö (engl. Integrated Development Environment)
ISM-taajuudet	Teollisuus, tiede ja lääketiedetaajuudet (engl. (Industrial, Scientific and Medical-Frequencies)
I^2C	Integroitujen piirien välinen yhteys (engl. Inter-Integrated Circuit)

L2CAP	Loogisten linkkien ohjaus- ja muokkaamisprotokolla (engl. Logical Link Control and Adaptation Protocol)
LMP	Linkkienhallintaprotokolla (engl. Link Management Protocol)
PDU	Protokolladatayksikkö (engl. Protocol Data Unit)
PIN	Henkilökohtainen tunnistenumero (engl. Personal Identifiaction Number)
PSRAM	Pseudostaattinen suorasaantimuisti (engl. Pseudostatic Random Access Memory)
RFCOMM	Radiotaajuusviestintäprotokolla (engl. Radio Frequency Communication)
SCO	Synkroninen yhdistymis-painotteinen linkki (engl. Synchronous Connection-Oriented Link)
SDK	Ohjelmistokehitystyökalu (engl. Software Development Kit)
SDP	Synkroninen yhdistymis-painotteinen linkki (engl. Service Discovery Protocol)
SMP	Turvallisuudenhallintaprotokolla (engl. Security Manager Protocol)
SPI	Sarjamuotoinen oheislaiteväylä (engl. Serial Peripheral Interface)
SPP	Sarjaporttiprofiili (engl. Serial Port Profile)
SRAM	Staattinen suorasaantimuisti (engl. Static Random Access Memory)
UUID	Yleinen ainutlaatuinen tunniste (engl. Universally Unique Identifier)

1. JOHDANTO

Bluetoothista on vuosien aikana tullut monille kuluttajille tuttu käsite, sillä tämä langattomaan tiedonsiirtoon kehitetty tekniikka on sulautunut meidän jokapäiväiseen elämäämme erittäin vahvasti. On harva se päivä, että emme käyttäisi sitä edes jossain muodossa, sillä esimerkiksi älykellot, langattomat kuulokkeet ja fitness-rannekkeet ovat yleistyneet huomattavasti viime vuosien aikana. Moni ei kuitenkaan tiedä tarkemmin mitä kaikkea tapahtuu taustalla, kun kahden Bluetooth-laitteen välille muodostetaan yhteys.

Tämän työn tarkoituksena on tutkia, miten Bluetooth sekä sen energiasäästeliäämpi rinnakkaisteknologia Bluetooth Low Energy on toteutettu. Toteutusta tutkitaan käyttämällä hyväksi ESP32-mikrokontrolleria, josta löytyy tuki kummallekin, klassiselle Bluetoothille ja Bluetooth Low Energylle.

Työn toisessa luvussa keskitytään työn aikana käytössä olevaan Firebeetle ESP32-mikrokontrollerin toiminnallisuuteen. Se on yksi useista markkinoilta saatavista mikrokontrollereista, jotka hyödyntävät tärkeimpänä komponenttinaan ESP32-moduulia. Kyseisen moduulin tärkeimpänä ominaisuutena onkin tuki Bluetoothin versiolle 4.2, jota tässä työssä tutkitaan. Luvussa käydään läpi ESP32:n tärkeimmät ominaisuudet laitteistopohjalta sekä sivutaan myös erilaisia tapoja, joilla kyseistä mikrokontrolleria pystyy ohjelmoimaan.

Kolmas luku käsittelee Bluetooth- ja Bluetooth Low Energy- teknologioiden teknillistä toteutusta sekä sitä, miten nämä kaksi teknologiaa eroavat toisistaan. Luvussa kerrotaan myös teknologioiden protokollapinojen eroista ja niiden lohkojen perustoiminnallisuudesta.

Neljännessä luvussa keskitytään Bluetooth-teknologioiden toteutukseen käytännön näkökulmasta. Luvussa kerrotaan, miten pystytään luomaan yhteys kahden laitteen välille, mitä se vaatii ESP32:lta ja miten pystytään lähettämään ja vastaanottamaan dataa Bluetooth BR/EDR- ja BLE -teknologioiden avulla.

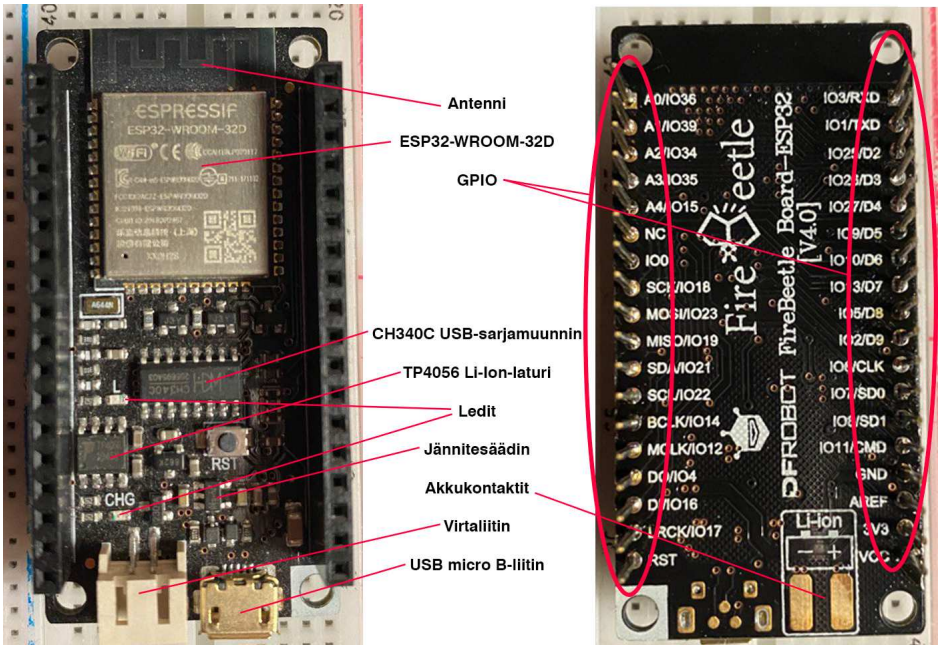
Viidennessä luvussa tehdään yhteenveto työn aikana käydyistä asioista ja käsitellään työn aikana tehtyjä havaintoja. Havaintojen pohjalta mietitään myös mahdollista jatkotutkimusta, jota voitaisiin tehdä aiheesta.

2. ESP32

ESP32 on shanghailaisen Espressif Systemsin valmistama mikrokontrollerijärjestelmä. ESP32:n tärkeimpiä tuettuja ominaisuuksia ovat sen tukemat 2.4 GHz Wi-Fi, Bluetooth ja energiatehokas Bluetooth-yhteys eli BLE (engl. Bluetooth Low Energy). ESP32 on noussut suureen suosioon pienten kotiprojektien tekijöiden sekä isompien yritysten ja tieteenharjoittajien keskuudessa. [17, Luku 1]. Suurimpina syinä tähän ovat etenkin valmistajan itse tekemä laaja dokumentaatio, tuki Bluetoothille ja Wi-Fille sekä laaja yhteisö, joka on tuottanut useita avoimessa jaossa olevia projekteja ja ohjeita ESP32:n käyttöön ja ohjelmointiin.

2.1 Tekniset tiedot

Tässä projektissa keskitytään etenkin ESP32-WROOM-32D-moduulilla varustettuun Firebeetle ESP32:n toimintaan, jonka voi nähdä kuvasta 2.1. Kuvaan on myös merkittyinä sen oleellisimmat komponentit, joita ovat ESP32-WROOM-32D-moduuli, USB-sarjamuunnin CH340C, Li-ion-laturi TP4056 sekä jännitteensäädin RT9080-33GJ5. [11, s. 1]



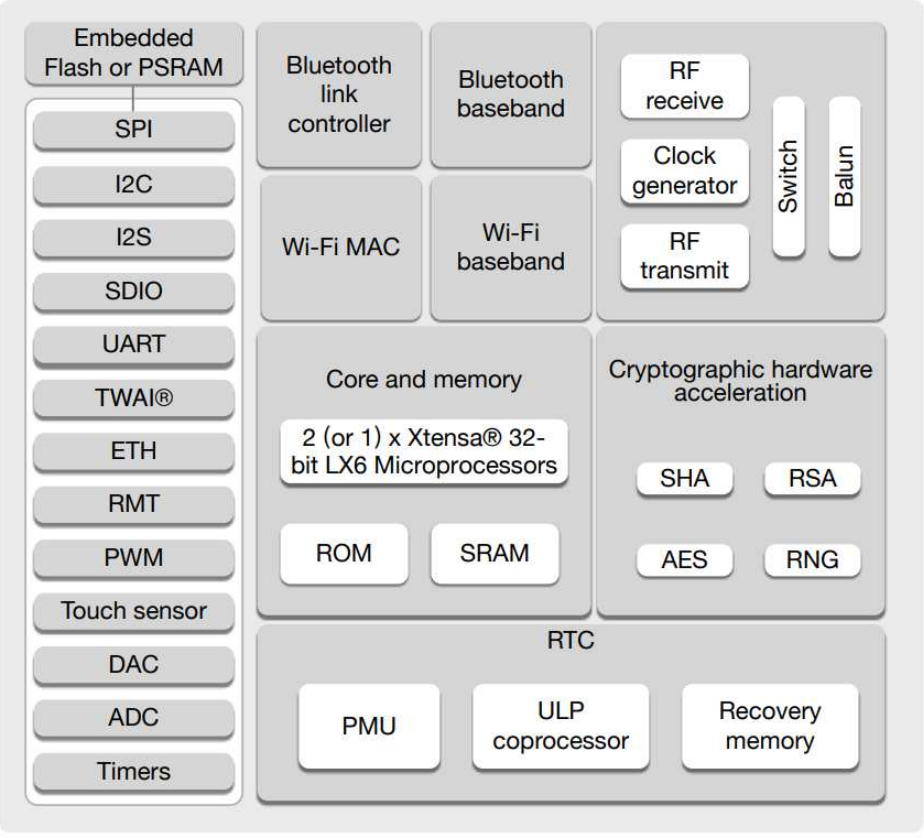
Kuva 2.1. *Firebeetle ESP32 ja sen oleellisimmat komponentit*

Firebeetlen tekniset tiedot löytyvät taulukosta 2.1. Kyseisen moduulin lisäksi Espressif valmistaa useita eri parametreilla varustettuja moduuleita. Nämä moduulit voivat erota käyttäjän tarpeiden mukaan esimerkiksi muistin määrältä, lämmönsiedolta sekä tuetuilta liittimiltä [17, Luku 1].

Taulukko 2.1. *Firebeetle ESP32:n tekniset tiedot [10][11]*

Ominaisuus	Arvo
Prosessori	2 ytiminen Tensilica Xtensa LX6 32-bit
Prosessorin kellotaajuus	säädettävä 80–240 MHz
SRAM	520 KB
Flash-muisti	16 Mbit
Toimintajännite	3.3 V
Tuetut väylät	I2C, I2S, SPI
Tuetut teknologiat	WI-FI (802.11 b/g/n), Bluetooth 4.2 (BR/EDR/BLE)
GPIO-Portit	9x digitaalista / 5x analogista

ESP32-D0WD-siru, johon ESP32-WROOM-32D pohjautuu, on jaettu yhdeksään eri toiminnalliseen lohkoon. Nämä lohkot voi nähdä kuvasta 2.2 Lohkojen Bluetooth link controller- ja Bluetooth baseband- toimintaan perehdytään vielä tarkemmin luvuissa 3 ja 4.



Kuva 2.2. *ESP-32 lohkokaavio [9, s. 12]*

Kyseisestä lohkokaaviosta poiketen ESP32-D0WD ei sisällä sisäänrakennettua Flash- eikä pseudostaattista suorasaantimuistia (engl. Pseudostatic Random Access Memory, PSRAM). Siitä syystä ESP32-WROOM-32D moduuliin on lisätty ulkoinen neljän megatavun muisti. Ulkoinen muisti on kytkettynä GPIO-, eli yleisiin sisään-/ulostuloportteihin (engl. General purpose input output) GPIO6-GPIO11. Tämän takia näitä portteja ei pysty käyttämään mikrokontrollerin tasolla [10, s. 8–9, 12]. Myös muissa GPIO-porteissa on niihin kytkettyjä ominaisuuksia, kuten analogiadigitaalimuunnin (engl. Analog to Digital Converter, ADC), digitaalianalogiamuunnin (engl. Digital to Analog Converter, DAC), kapasitiivisen kosketuksen tunnistus, pulssinleveysmodulaattorit sekä I2C- ja sarjamuotoisen oheislaiteväylän (engl. Serial Peripheral Interface, SPI) liittimet. Tästä syystä ennen projektin aloittamista ESP32:ta käyttäen, täytyy tutustua dokumentaatioon ja selvittää, mitkä tarvittavista ominaisuuksista ovat missäkin portissa ja mitä portteja voi käyttää tavallisina analogisina/digitaalisina sisään- ja ulostuloportteina. [9, s. 33–43]

2.2 ESP32-Ohjelmointityökalut

Suuri suosio on taannut ESP32:lle laajan yhteisön taakseen. Tämä on johtanut siihen, että käyttäjät ovat vuosien varrella luoneet lukuisia rajapintoja ESP32:n ohjelmoimiseen. Koska ESP32:n tuomat mahdollisuudet ovat erittäin laajat, eri käyttäjillä on erilaiset vaatimukset toiminnallisuudelle.

ESP32 tukee toisella suositulla mikrokontrollerilla Arduino Unolla käytettyä Arduino- integroitua ohjelmointiympäristöä (engl. Integrated Development Environment, IDE). Sen avulla myös vähemmän ohjelmoivat ihmiset ja aloittelijat pääsevät luomaan yksinkertaisia projekteja. Arduino IDE:ssä on kirjastot yleisimmille markkinoilla oleville komponenteille, kuten esimerkiksi nestekidenäytöille, lämpötila- ja kosteusantureille, servomoottoreille sekä ESP32:n sisäisille ominaisuuksille kuten AD-muuntimille ja Hall-sensorille. Haittapuolena kyseisen ympäristön kanssa on kuitenkin se, että se on rajattuna pelkästään näihin kirjastoihin. Jos haluaa luoda mitään monimutkaisempaa, niin täytyy siirtyä toisten rajapintojen käyttöön. [17, luku 1]

Etenkin web-ohjelmoijille tutun JavaScriptin käyttö onnistuu käyttämällä Moddable- ohjelmistokehitystyökalua (engl. Software Development Kit, SDK). Se käyttää XS-nimistä JavaScript-moottoria, jonka avulla pyritään pienentämään web-ohjelmoinnissa vähemmälle huomiolle jäävät ongelmat muistinhallinnan kanssa sekä optimoimalla koodin suoritusta kääntämällä JavaScriptin mikrokontrollerille nopealukuisemmaksi tavukoodiksi [14]. Moddable SDK:n tärkeimpiä ominaisuuksia ovat tuki web-ohjelmoinnissa käytettäville ohjelmatason protokollille sekä useammalle graafiselle kirjastolle. [13]

ESP-IDF on Espressifin itse luoma rajapinta ohjelmoimiseen, jossa kielenä käytetään C:tä. Se tukee kaikkia yleisimpiä käyttöjärjestelmiä ja sen voi asentaa joko itsenäisenä komentorivityökaluna tai lisäosaksi VS Code- ja Eclipse- ohjelmointiympäritöihin. ESP-IDF sisäl-

tää useita eri ohjelmointirajapintoja (engl. Application programming interface, API), jotka tarjoavat tuen kaikille ESP32:lla toteutettavissa oleville ominaisuuksille, kuten esimerkiksi muistinhallinnalle, ADC-muuntimille, verkkoprotokollien toteutuksille ja sisäänrakennetuille sensoreille [6].

Kyseisen työn kannalta tärkeimmät ohjelmointirajapinnat ovat kuitenkin Bluetooth ohjelmointirajapinnat, joita ESP-IDF tukee kahta. Näistä vakiona käytetään Bluedroid-nimistä rajapintaa, jonka avulla pystyy ohjelmoimaan sekä yleistä Bluetooth-, että BLE-teknologioiden mukaista toiminnallisuutta. Toinen on pelkästään BLE:tä varten tarkoitettu nimBLE, joka eroaa Bluedroidista paremmalla muistinhallinnalla. [7]

3. BLUETOOTH

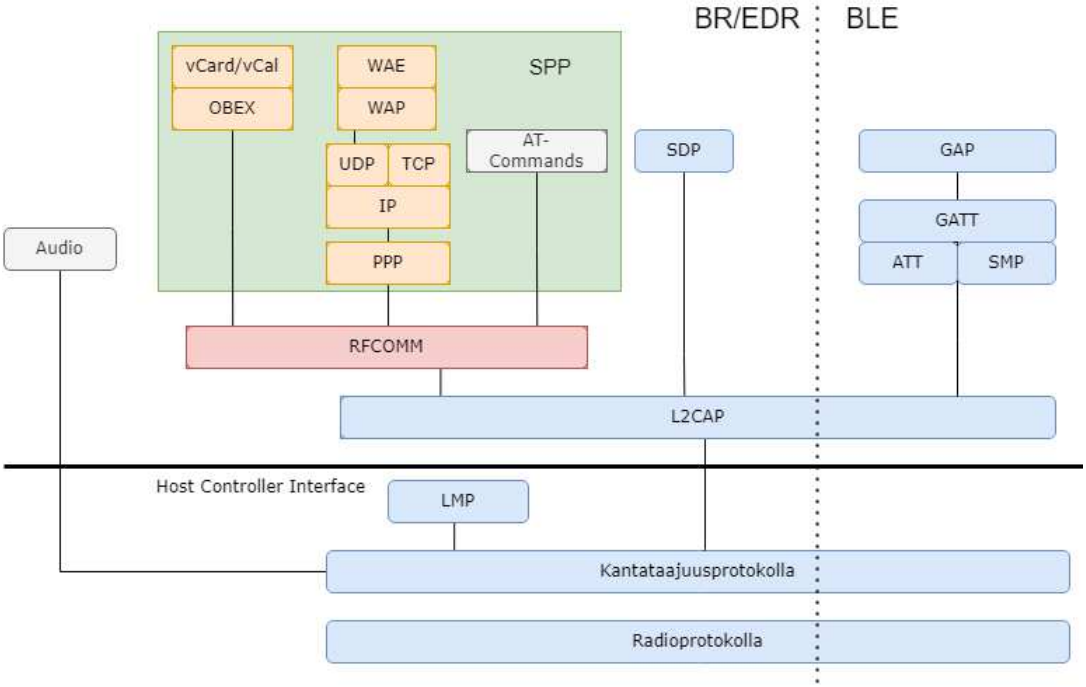
Bluetooth on vuonna 1994 ruotsalaisen L. M. Ericssonin kehittämä langattoman tiedonsiirron tekniikka, joka on saanut nimensä 940–981 eaa. eläneen Tanskan kuninkaan Harald Sinihampaan (engl. Bluetooth) mukaan [21, s. 1]. Bluetooth-tekniikan standardien ylläpitämisestä vastaa The Bluetooth Special Interest Group (Bluetooth SIG), jonka perustivat vuonna 1998 Ericsson, Intel, IBM, Nokia ja Toshiba. Tällä hetkellä Bluetooth SIG:n kuuluu yli 36 000 yritystä, jotka käyttävät tai kehittävät Bluetooth-standardia [3]. Bluetoothin alkuperäinen idea oli luoda avoin standardi lyhytkantamataajuusteknologialle, jolla pystyttäisiin korvaamaan fyysinen linkki tietokoneiden välillä. Vuosien aikana Bluetooth-standardia on kehitetty esimerkiksi nostamalla tiedonsiirtonopeutta sekä nostamalla kantamaa teoreettisesti jopa 1.6 kilometriin vuonna 2016 julkaistussa versiossa 5.0 [1, s. 139]. ESP32:n tukema Bluetooth-versio on vuonna 2014 julkaistu 4.2.

Nykyään kaksi Bluetoothin yleisintä käytössä olevaa teknologiaa ovat tavallisen tiedonsiirtonopeuden/parannetun tiedonsiirtonopeuden BR/EDR (engl. Basic data rate/Enhanced Data Rate)-teknologia, jota sanotaan myös klassiseksi Bluetoothiksi sekä energiatehokas Bluetooth-yhteys BLE (engl. Bluetooth Low Energy). Näiden lisäksi on myös olemassa Bluetooth Mesh-verkko, jonka avulla pystytään luomaan laitteiden välisiä verkostoja, joissa laitteet pystyvät tilaamaan lähetettävän datan julkaisijalaitteilta. Sen toteutukseen ei tässä työssä kuitenkaan perehdytä sillä se on rakennettu BLE-spesifikaation päälle. [21, s. 3, 8–9]

3.1 Bluetooth BR/EDR-protokollat

Bluetooth käyttää hyväkseen vapaassa käytössä olevia ISM (Industrial, Scientific and Medical)-taajuuksia väliltä 2400–2483.5 MHz, jotka on ensisijaisesti tarkoitettu korkeataajuustekniikoita hyödyntäville laitteille ja joita ei olla alun perin luotu radiolaitteiksi [20, s. 41] [18]. ESP32 tapauksessa Bluetooth-pino koostuu kymmenestä eri protokollasta ja profiilista. BR/EDR ja BLE-teknologioille yhteisiä protokollia ovat kantataajuusprotokolla, radioprotokolla, linkkienhallintaprotokolla LMP (engl. Link Management Protocol) ja loogisten linkkien ohjaus- ja muokkaamisprotokolla L2CAP (engl. Logical Link Control and Adaptation Protocol). Palvelunlöytämisprotokolla SDP (engl. Service Discovery Protocol), radiotaajuusviestintäprotokolla RFCOMM (engl. Radio Frequency Communication) sekä

sarjaporttiprofiili SPP (engl. Serial Port Profile) ovat toteutettu vain BR/EDR:lle. BLE:n toimintaa ohjaavat yleinen pääsyprofiili GAP (engl. Generic Access Profile), yleinen ominaisuusprofiili GATT (engl. Generic Attribute Profile) sekä turvallisuudenohjausprotokolla SMP (engl. Security Manager Protocol). Näiden sijoitukset lohkokaaviossa pystyy näkemään kuvasta 3.1. [8]



Kuva 3.1. *Bluetooth protokollapino [21, s. 5] [12, s. 144]*

Kuvaajassa näkyvien protokollien lisäksi ESP32:lla on myös toteutettuna edistynyt audionvälitysprofiili A2DP (engl. Advanced Audio Distribution Profile) ja audio/video etäohjausprofiili AVRCP (engl. Audio/Video Remote Control Profile), joiden avulla pystytään välittämään audio- sekä videodataa Bluetoothin avulla. Nämä kaksi protokollaa sijoittuisivat GAP:n sisälle [8, s. 15–17].

3.1.1 Fyysisen kerroksen protokollat ja HCI

Radioprotokolla vastaa signaalinkäsittelystä muokkaamalla lähetettävän datan Gauss-suodatetun vaihtotaajuusmodulaation (engl. Gaussian Frequency-Shift Keying, GFSK) mukaisesti symboleihin. GFSK on paranneltu versio vaihtotaajuusmodulaatiosta (engl. Frequency Shift Keying, FSK), jossa symbolit ovat asetettuna joihinkin kantoaallon taajuuksiin. GFSK:n ero FSK:hon on se, että ennen modulaatiota se vielä laitetaan Gauss-suodattimen läpi, jolla saavutetaan pienempi spektrin leveys. Spektrin leveyden pienentäminen vähentää ei-toivotuille sivukaistoille menevää tehoa, minkä ansiosta hyödyllisen signaalin lähetysteho kasvaa. Muokkauksen jälkeen radioprotokolla lähettää symbolit Bluetoothin käyttämälle taajuuskaistalle, joka on välillä 2400–2483.5 MHz. Kyseinen

kaista on jaettu 79 erilliseen 1 MHz levyiseen kanavaan taajuustasossa ja 625 mikrosekunnin pituisiin lohkoihin aikatasossa, mikä antaa yhteensä 1600 lohkoa sekunnissa datan lähettämiseen. Näin saadaan aikaan Bluetooth 1.0 standardin mukainen Basic Rate-toiminnallisuus, joka mahdollistaa 732.2 kb/s datansiirtonopeuden. Tämä on Bluetooth standardilta vaadittu ominaisuus, jota on seuraavissa versioissa laajennettu muilla lohkoilla. [21, s. 4]

Kantataajuusprotokollan tehtävänä on vastata lähetetyn ja vastaanotetun datan prosessoinnista luomalla kantataajuus-paketteja, tekemällä niihin virheenkorjaukset, hallinnoimalla laitteiden välisiä linkkejä sekä tekemällä tietovuonohjausta. Tällä tasolla on myös toteutettu ero BR/EDR ja BLE-teknologioiden välillä. Kummallekin teknologialle toteutetaan oma pakettirakenne. Kuten radioprotokolla, myös kantataajuusprotokolla kuuluu Basic rate -toiminnallisuuteen ja on pakollinen osa myös uudempia Bluetoothin versioita. [4, s. 375–376]

Host Controller Interface eli isäntäohjainrajapinta on edellä mainittujen kantataajuusprotokollan, radioprotokollan ja linkkienhallintaprotokollan yhdistävä rajapinta. Sen avulla ylemmän kerroksen sovellukset pystyvät hallinnoimaan Bluetoothin fyysisen tason protokollia yhden standardoidun rajapinnan kautta [15, luku 5]. Tämä mahdollistaa sen, että Bluetooth-pinon toteutus ei ole valmistajariippuvainen, vaan radioyksikkö voidaan hankkia erikseen ja sen päälle sitten rakentaa sovelluskerroksen toteutus. HCI ei kuitenkaan ole pakollinen, jos koko protokollapino on toteutettu samalla prosessorilla [12, s. 62].

3.1.2 LMP

LMP eli linkkienhallintaprotokolla operoi suoraan kantataajuusprotokollan kanssa ja hallinnoi kahden laitteen välisen yhteyden muodostamista sekä laitteiden välisten yhteyksien ohjausta. Kaikki tällä tasolla kulkevat paketit eivät etene ylemmille tasoille ja ovat käytössä vain kummankin laitteen linkkienhallintalohkoissa. [4, s. 538]

Linkkienhallintaprotokollan tehtäviin kuuluvat myös laitteiden todentaminen ja niiden välisen linkin salaus. Näistä todentaminen on pakollinen osa Bluetooth-standardia. Autentikaatio tapahtuu ennestään muodostetussa yhteydessä lähettämällä 16-tavuinen satunnaisluku, jonka perusteella vastaanottaja pystyy toteamaan, onko koodin lähettänyt laite todennettu vastaanottajassa. Jos linkkiä ei ole ennestään muodostettu, niin autentikaatio tapahtuu henkilökohtaisen tunnuslukukoodin (engl. Personal Identification Number, PIN) pohjalta rakennetun väliaikaisavaimen avulla. [15, Luku 5]

Bluetooth tukee kahta eri linkkityyppiä, jotka ovat synkroninen yhdistymis-painotteinen SCO (engl. Synchronous Connection-Oriented) ja asynkroninen yhdistymis-painotteinen ACL (engl. Asynchronous connection-oriented). SCO on tarkoitettu pitkäaikaisten ja jatkuvien linkkien muodostamiseen ja on käytössä esimerkiksi audiolaitteiden kanssa. ACL

taas on pakettipohjainen lähetyslinkki, jossa tieto välitetään pienissä lohkoissa sekä nimensä mukaan asynkronisesti. Tämä mahdollistaa useampien yhtäaikaisten ACL-linkkien muodostamisen. [12, s. 46]

3.1.3 L2CAP ja SDP

L2CAP eli loogisten linkkien ohjaus- ja muokkaamisprotokolla toimii linkkinä ylempien protokollatasojen ja fyysisen kerroksen protokollien välillä. Se koostuu lopullisesta laitteesta riippuen joko BR/EDR-ohjaimesta, BR/EDR/LE-ohjaimesta tai pelkästä LE-ohjaimesta.

L2CAP:n tärkeimpiä ominaisuuksia ovat [4, s. 1721–1723]:

- **Protokolla/kanavamonikertaistaminen ja vuonohjaus** L2CAP pystyy hoitamaan useampien ylemmän tason protokollien samanaikaisen toiminnan tukemalla useampaa samanaikaista kanavaa. Jokaiselle kanavalle asetetaan oma vuonohjaus, jotta fyysisen tason protokollat eivät ylikuormittuisi.
- **Paloittelu ja uudelleenkokoaminen** Ylempien protokollatasojen paketit ovat suurempia, kuin fyysinen taso pystyy lähettämään. L2CAP-taso paloittelee saadun datan pienempiin L2CAP-protokolladatayksikköpaketteihin (engl. Protocol Data Unit, PDU), minkä avulla pystytään hallinnoimaan mahdollisia viiveitä pakettikokoja muuttamalla, saavuttamaan parempi virheensieto ja voidaan helpottaa muistinhallintaa.
- **Virheenhallinta ja uudelleenlähetys** L2CAP hoitaa virheellisten pakettien uudelleenlähetysten hallinnan yhteistyössä vuonohjauksen kanssa. Virheenhallinta tapahtuu L2CAP-paketin otsakkeessa olevalla kehyksen tarkastusjaksolla (engl. Frame Check Sequence, FCS), jossa paketin hyötykuormasta lasketaan FCS-arvo. Kyseistä arvoa verrataan vastaanotetusta hyötykuormasta laskettuun FCS-arvoon ja saadaan selville, onko lähetyksen aikana tullut bittimuutoksia.

Palvelunlöytämisprotokollan eli SDP:n avulla Bluetooth-laitteet pystyvät tunnistamaan toistensa tarjoamat palvelut. Tämä tapahtuu yksinkertaisella kysymysvastaus L2CAP-paketilla, missä palveluita tiedusteleva SDP-asiakas lähettää toiselle laitteelle kyselyn ja saa vastauksena listan vastaanottavan laitteen tarjoamista palveluista. Kun asiakas päättää ottaa jonkun palveluista käyttöön, luodaan uusi yhteys sitä hoitavaan protokollaan. [19, s. 5]

3.1.4 Profiilit ja GAP

Kaikki käyttäjälle näkyvä Bluetoothin toiminnallisuus toteutetaan profiilien avulla. Profiilit ovat malleja, joiden avulla pystytään määrittämään ja toteuttamaan Bluetooth-laitteen toiminnallisuus [15, Luku 3]. Tällainen ennalta määritetty toimintojen profilointi mahdollistaa esimerkiksi sen, että pystytään takaamaan toiminta eri valmistajien laitteiden välillä ilman mitään ylimääräisiä asetuksia. [12, s. 331].

Yleinen pääsyprofiili GAP on kaikista profiileista tärkein. Se määrittää kaikille Bluetooth-laitteille pakolliset pinon osat, jotka ovat BR/EDR tapauksessa edellä mainitut SDP, L2CAP, LMP, kantataajuusprotokolla ja radioprotokolla. GAP:n kautta myös hallinnoidaan laitteiden osoitteita, määritetään laitteen nimi esimerkiksi yhdistettävien laitteiden luettelossa puhelimella, hallinnoidaan saman laitteen kanssa uudelleenyhdistäminen ja määritetään laitteen rooli Bluetooth yhteydessä [12, s. 103–104].

BR/EDR-tilassa rooleja on kahta erilaista. Nämä ovat A-osakas sekä B-osakas. A-osakas on se, joka aloittaa uusien yhteyksien etsimisen, muodostaa uusia yhteyksiä sekä hallinnoi jo tuttujen linkkien uudelleenyhdistämisen. B-osakas toimii vastaavasti yhteyden vastaanottajana. Nämä roolit eivät ole kuitenkaan poissulkevia ja perinteisemmässä BR/EDR Bluetooth-yhteydessä laite voi toimia sekä A-osakkaana, että B-osakkaana [4, s. 1978, 1981–1982].

Roolien asettamisen lisäksi yleisellä pääsyprofiililla määritetään laitteen näkyvyys ja yhdistettävyys muille lähistöllä oleville Bluetooth-laitteille. Näitä tiloja ovat havaittava (engl. discoverable), ei havaittava (engl. non-discoverable), yhdistettävä (engl. connectable) sekä ei yhdistettävä (engl. not-connectable). Havaittavissa olevalle laitteelle löytyy myös kaksi eri tyyppiä: yleisesti havaittava (engl. general discoverable) ja rajoitetusti havaittava (engl. limited discoverability). Yleisesti havaittavalle laitteelle kaikki lähellä olevat laitteet pystyvät tekemään yhteydenmuodostuspyynnön, kun taas rajoitetusti havaittavan laitteen yhteydenmuodostuksessa on rajoitteita, kuten rajoitettu yhdistysaika tai vaatimus jonkin ehdon täyttymiselle. [12, s. 105] [4, s. 1993–1997]

3.1.5 RFCOMM ja SPP

Koska Bluetoothin idea oli korvata fyysinen tiedonsiirtolinkki laitteiden välillä ja Bluetooth-standardin kehittämisen aikana useat laitteet käyttivät datan välitykseen sarjaliikennettä, päätettiin sen korvaava toiminnallisuus lisätä myös Bluetooth-standardiin. Näin syntyi radiotaajuusviestintäprotokolla RFCOMM. RFCOMM-yhteydessä kahden laitteen välille muodostetaan linkit käyttämällä L2CAP:ia, jonka avulla on mahdollista muodostaa jopa 60 samanaikaista sarjaliikennelinkkiä. Toisin kuin fyysisissä sarjaliikenneliittimissä, RFCOMM:lla ei ole useaa signaalia käytössä. Tämän takia sarjaliikenteelle tyypilliset ominaisuudet kuten signaalien ajoitus ja laitteiden välinen ohjauskommunikointi tapahtuu käyttämällä komentoja ja vastauksia niihin. Ajoitus sen sijaan tapahtuu kokonaan fyysisen tason protokollien ohjaamana, toisin kuin langallisessa sarjaliikenteessä. Siinä ajoituksesta vastaavat laitteiden kellot ja niiden välinen synkronointi [15, Luku 8].

RFCOMM:n päälle rakentuu myös yksi oleellisimmista Bluetoothissa olevista profiileista, joka on sarjaporttiprofiili SPP. Sarjaporttiprofiilin tehtävä on määrittää ja alustaa radiotaajuusviestintäprotokollaa käyttävä virtuaalinen radiolinkki laitteiden välille. Alun perin SPP:tä käytettiin suurimmaksi osaksi modeemeissa käytettyjen ohjauskomentojen lähet-

tämiseen muiden profiilien puolesta. SIG:n huomattua sen potentiaali myös tiedonsiirtoon, siihen lisättiin yleisen objektienvaihtoprofiilin (engl. Generic Object Exchange Profile, GOEP) toiminnallisuus. Näin SPP:stä tuli pohja usealle myöhemmin kehitetylle profiilille, missä SPP toimii yleensä väliohjelmistona. [15, Luku 14]

3.2 Bluetooth Low Energy

BLE eli Bluetooth Low Energy on vuonna 2014 Bluetooth standardiin 4.0 tullut teknologia, joka eroaa perinteisestä BR/EDR:stä pienemmällä energiankulutuksella ja yksinkertaisemmalla protokollapinolla. Ensisijaiset käyttökohteet BLE-teknologialle ovat olleet alusta alkaen erilaiset anturit ja laitteet, joiden on tarkoitus lähettää pieniä määriä dataa useasti ja pystyä operoimaan pitkiä aikoja saman akun varassa [12, s. 133].

Pienempi energiankulutus saavutetaan BLE:ssä etenkin lähettämällä dataa pelkästään silloin, kun se on tarpeellista. Tämä perustuu siihen, että jatkuva radiolinkin ylläpitäminen vie erittäin paljon energiaa. Verrattuna tavalliseen BL/EDR-Bluetoothiin, BLE:ssä ei yritetä selvittää radiokanavan tilaa, vaan oletetaan, että päälaite kuuntelee lisälaitetta jo ennestään sovitulla kanavalla. Tämä mahdollistaa lisälaitteen toiminnan yksinkertaistamisen pelkkään lähettimeen [12, s. 138–141].

BR/EDR yhteydestä poiketen BLE käyttää vain 40 kanavaa, sillä kanavien välinen etäisyys on 2 MHz verrattuna 1 MHz BR/EDR-Bluetoothissa. Tämä helpottaa etenkin suodattimien toteutusta laitteissa, mikä vastaavasti edistää parempaa virrankulutusta [4, s. 1978, 171] [21, s. 6]. Käytössä olevista 40 kanavasta kolme on kuitenkin varattu pelkkään mainostamiseen. Erillisten data- ja mainostuskanavien lisäys mahdollistaa nopeamman yhdistämisen laitteiden välillä, sillä BLE:n tarvitsee etsiä laitteita vain näiltä kolmelta kanavalta. Näin saavutetaan jopa kolmen millisekunnin yhdistämisaika verrattuna BR/EDR-Bluetoothin 20 millisekuntiin. [12, s. 138]

3.2.1 GAP

BLE:ssä GAP:n avulla toteutetaan menettelyt laitteiden yhdistämisen, turvallisuuden ja löytämisen suhteen. Tämän lisäksi GAP määrittää protokollapinon pakolliset osat, joita ovat kaikki HCI:n hoitamat protokollat sekä L2CAP, ATT/GATT ja SMP. [12, s. 331] Myös BLE:ssä laitteet jaetaan tällä tasolla eri rooleihin, joiden mukaan laitteen toiminnallisuus rajoittuu. Näitä rooleja ovat [4, s. 253]:

- **Lähettäjärooli** Laite mainostaa itseään yhdistettäväksi laitteeksi lähipiirissä oleville laitteille.
- **Tarkkailijarooli** Laite etsii lähistöllä olevia muita laitteita, joihin se voisi muodostaa yhteyden.

- **Lisälaiterooli** Laite toimii slave-tilassa, eli pystyy lähettämään dataa master-laitteelle, mutta ei pysty muodostamaan uusia yhteyksiä muihin laitteisiin.
- **Keskeinen rooli** Laite toimii master-tilassa, eli antaa lisälaiteroolissa olevalle laitteelle käskyjä ja pystyy vastaanottamaan dataa useammasta eri lisälaiteroolissa olevalta laitteelta.

3.2.2 ATT ja GATT

Toisin kun BR/EDR-Bluetoothissa, BLE-pino ei sisällä SDP-protokollaa vaan se on sisäänrakennettu yleiseen ominaisuusprofiiliin GATT:iin. GATT toimii yleisenä rajapintana kahden BLE-laitteen välillä ja sen avulla pystyy hoitamaan linkin luomisen, datan välityksen, toisen laitteen ominaisuuksien selvittämisen sekä niiden lukemisen/kirjoittamisen. GATT:n välisissä yhteyksissä laitteet jakautuvat kahteen rooliin. Nämä ovat palvelin ja asiakas. Asiakas voi lähettää palvelimelle pyyntöjä, joihin palvelin vastaa sen tarjoamien ominaisuuksien perusteella antamalla esimerkiksi anturidataa tai luvan kirjoittaa dataa rekisteriinsä. [4, s. 2211–2214]

Kaikki tämä tapahtuu käyttämällä matalamman tason ominaisuusprofiileja (engl. Attribute Protocol, ATT). Nämä ovat yksinkertaisia protokollia, jotka ennalta määritettyjen arvojen avulla määrittää palvelimen ominaisuudet asiakkaalle. ATT-profiilit koostuvat sen tarjoamien ominaisuuksien tunnisteista, niihin liittyvistä luvista, kuten esimerkiksi kirjoituksia lukuluvat sekä profiilin yleisesti uniikista tunnisteesta (engl. Universally Unique Identifier, UUID). UUID:t ovat SIG:n ennalta määräämiä tunnisteita, joiden avulla asiakkaan ei tarvitse selvittää mitä palveluita on tarjolla. Riittää, että annetaan pyyntö jollekin tietylle UUID:lle ja jos sellainen löytyy palvelimelta, saadaan siihen vastaus. [4, s. 2164–2167]

3.2.3 SMP

Turvallisuudenohjausprotokolla SMP on oleellinen osa BLE:n toteutusta. Toisin kuin BR/EDR-Bluetoothissa, BLE:ssä ei ole linkkienhallintaprotokollaa, joka hoitaisi linkin turvallisuuden. SMP:n tehtävänä onkin hoitaa laitteiden välinen autentikaatio, muodostaa turvallinen yhteys laitteiden välille sekä salata niiden välinen tietoliikenne. [12, s. 231]

Yhteyden muodostaminen tapahtuu kolmivaiheisena. Ensimmäisessä vaiheessa tarkkailijaroolissa oleva laite voi kysyä lähettäjälaitteelta salaisen linkin muodostusta. Tähän lähettäjälaite vastaa yhteydenmuodostuspyynnöllä, jossa se tiedustelee tarkkailijalaitteen tarjoamia ominaisuuksia ja ehdottaa avaimia, joita tultaisiin käyttämään kolmosvaiheessa. Tarkkailija vastaa tähän kyselyyn lähettämällä vastausviestin, jolla se hyväksyy tarjotut avaimet ja lähettää tietoa sen tukemista ominaisuuksista. [12, s. 247–248]

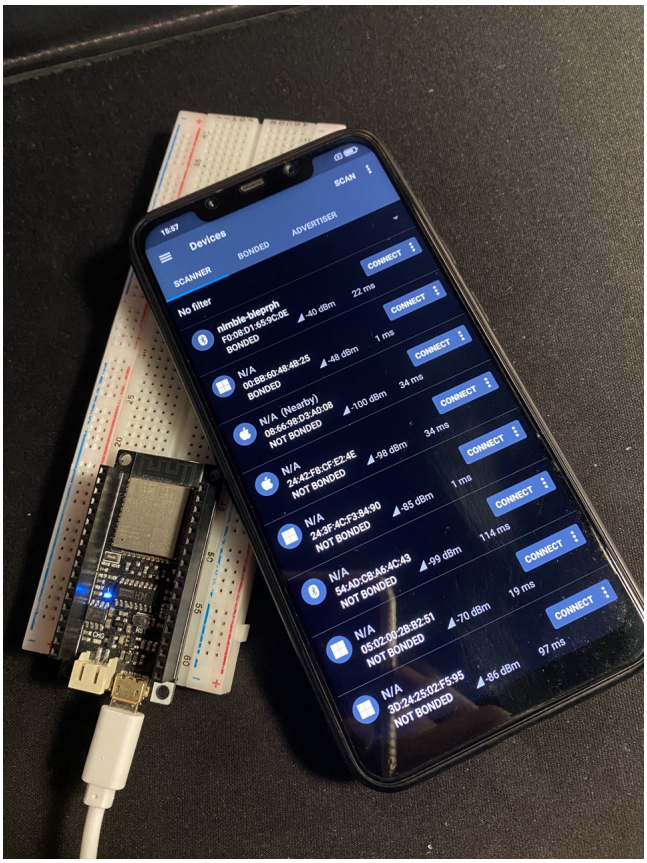
Toisessa vaiheessa tapahtuu avainten vaihto, missä lähettäjä luo 128-bittisen lyhytaikai-

sen avaimen ja lähettää tämän tarkkailijalle. Tarkkailija vastaavasti luo oman 128-bittisen avaimen ja lähettää sen lähettäjälaitteelle. Tämän jälkeen tulee avainten tarkistusvaihe, missä tarkkailija ja lähettäjä lähettävät satunnaiset luvut toisilleen. Niiden avulla ne laskevat vastausviestit ja todentavat, että avaimet toimivat. [12, s. 248–250]

Kolmas vaihe on vapaaehtoinen ja sen avulla voidaan suojata kaikki tuleva liikenne salaamalla se. Kyseisen vaiheen aikana joko tarkkailija tai lähettäjä lähettää kaikki salaamiseen vaadittavat tiedot toiselle osapuolelle, jolloin näiden perusteella pystytään luomaan salattu tietoväylä. [12, s. 251–252]

4. KÄYTÄNNÖN TOTEUTUS

Tässä osiossa keskitytään siihen, kuinka ESP-IDF:n avulla pystytään toteuttamaan ohjelmistototeutus BR/EDR- ja BLE-teknologioiden toimimiseksi. ESP32 kiinnitetään tietokoneeseen USB-kaapelilla, jonka avulla pystytään välittämään tietokoneella kirjoitetusta koodista käännetyt binäärit ESP32:lle ilman erillistä ohjelmointityökalua. Kuvasta 4.1 voi nähdä käytettävän laitteiston.



Kuva 4.1. *Käytettävä laitteisto*

Kummassakin esimerkissä vastaanottajalaitteena toimii Android-käyttöjärjestelmällä toimiva Pocophone F1-puhelin. Puhelimelle on asennettuna sovellukset, joiden avulla pystytään havainnoimaan Bluetoothin toimivuus.

4.1 Bluetooth SPP-palvelin

BR/EDR Bluetoothin toimintaa tarkastellaan ESP-IDF:n tarjoaman bt_spp_acceptor- esimerkin avulla [5]. Esimerkissä ESP32 toimii sarjaporttiprofiilia käyttävänä palvelimena, johon muodostetaan salattu yhteys käyttämällä PIN-koodia. ESP32 on USB-kaapelilla yhdistettynä tietokoneeseen, jolla koodi käännetään ja kirjoitetaan laitteelle. SPP-palvelimen toimintaan liittyvien funktioiden toteutukset ovat liitteissä A ja B.

Esimerkin main-funktio koostuu seitsemästä funktiosta, joiden tehtävänä on alustaa ja käynnistää Bluetooth-toiminta ESP32:lla ja tämän jälkeen toteuttaa profiilien toiminnallisuus. Näiden funktioiden palautusarvoina toimii enum, eli luettelotyyppinen `esp_err_t`. Luettelotyyppi on ennalta määritetyistä vakioalkioista muodostettu lista, jonka avulla pystytään esimerkiksi lisäämään koodin luettavuutta antamalla taikaluvuille nimet [16, Luku 18]. Kyseisten esp_err_t arvojen avulla ohjelmoija pystyy helposti määrittämään virheen tyypin, joka asetetaan, kun ohjelman suorituksessa tapahtuu virhe. Tässä esimerkissä käytetään listan arvoa ESP_OK, jolla todennetaan, että Bluetoothin asetusvaiheessa ei ole syntynyt virheitä.

Ennen koodin kääntämistä ESP32 täytyy konfiguroida. ESP-IDF tarjoaa tähän terminaalisovellusta käytettäessä menuconfig-komennon tai graafisen käyttöliittymän, jos käytössä on ESP-IDF:n VS Code-lisäosa. Bluetooth-toiminnallisuuden tapauksessa menuconfigista pystyy esimerkiksi määrittämään, käytetäänkö BR/EDR vain BLE:tä, montako samanaikaista linkkiä voidaan muodostaa, asettamaan käytettävän Bluetooth-rajapinnan sekä asettamaan profiilit, joita tullaan käyttämään koodissa. Näiden konfiguraatioiden perusteella koodin kääntämistä pystytään optimoimaan käyttämällä näihin parametreihin sidottuja `#if-#endif` -rakenteita. Kun kääntäjälle tulee vastaan tällainen rakenne ja siinä oleva ehto ei täyty, kääntäjä ei ota sitä huomioon ja poistaa sen lopullisesta käännetystä koodista [16, Luku 23]. Näin saadaan säästettyä muistia, jota ei mikrokontrollereilla ole paljoa ylimääräistä.

4.1.1 Bluetooth-pinon alustus ja käynnistys

Ensimmäisenä tehdään Bluetooth-ohjaimen alustus. Se tapahtuu kutsumalla funktiota `esp_bt_controller_init(&bt_cfg)` Funktio ottaa parametrikseen ESP32 konfiguraatiorakenteen, jonka sisältö määritettiin menuconfig-komennon avulla. Alustusfunktion tehtävänä on allokoida tehtävät ja muut resurssit Bluetooth-pinon käyttöön. Se myös asettaa Bluetoothille keskeytykset, joiden avulla kaikki Bluetooth toiminnallisuus toteutetaan. Tässä vaiheessa voi huomata, että kaikki Bluetooth-ohjaimen kooditoteutus ei ole nähtävissä ESP-IDF:n käyttäjälle, vaan se on tallennettuna suoraan binäärimuodossa projektin tiedostoihin ja sen toiminnallisuus on saatavilla vain rajapintafunktioiden avulla.

Seuraavassa vaiheessa ohjain käynnistetään funktiolla `esp_bt_controller_enable()`.

Funktiolle annetaan parametrina tila, jossa ohjain aiotaan käynnistää. Tässä esimerkissä käytetään ainoastaan BR/EDR Bluetoothia, joten parametriksi annetaan sitä vastaava enum-tyyppi. Käynnistyksessä asetetaan päälle fyysisen kerroksen protokollat kutsumalla funktiota `esp_phy_enable()` ja `btdm_check_and_init_bb()`, joka ottaa parametrikseen käynnistettävän tilan tyypin.

Kun ohjain on alustettu, täytyy myös alustaa Bluedroid-rajapinta. Bluedroid on ESP-IDF:n vakiona tukema pinojärjestäjä, jonka tehtävä on luoda Bluetooth ja BLE standardien mukainen toteutus ESP32:lle [17, Luku 8]. Tässä vaiheessa etsitään sekä yhdistytään ESP:n isäntäohjainrajapintaan, jonka avulla pystytään kommunikoimaan fyysisen tason protokollien kanssa.

Alustuksen jälkeen Bluedroid käynnistetään lähettämällä ohjaimelle käynnistyskäsky. Kommunikointi Bluetooth-ohjaimen kanssa tapahtuu funktiolla `btc_transfer_context(btc_msg_t *msg, void *arg, int arg_len,btc_arg_deep_copy_t copy_func)`. Ensimmäinen parametri on viesti, jolla ilmoitetaan käskyn lähettäjäprofiilin tunniste, käskyn tyyppi ja suunta. Käskyn suunnalla tarkoitetaan sitä, meneekö käsky sovellukselta ohjaimelle vai toisinpäin. Toisella parametrilla välitetään mahdolliset argumentit, jotka halutaan lähettää käskyn mukana. Kolmas parametri on argumenttien koko tavuina ja neljäs on osoitin funktioon, jolla luodaan syväkopio edellisissä parametreissa annetuista viestistä ja argumenteista. Syväkopion avulla voidaan varmistaa, että kyseiset parametrit jäävät myös ESP32:n muistiin. Kun viesti ja argumentit on lähetetty ohjaimelle, se vuorostaan lähettää ne tarkkailijalaitteelle.

4.1.2 Profiilien alustus ja käynnistys

Tässä esimerkissä käytetään kahta Bluetooth-profiilia. Nämä ovat yleinen pääsyprofiili GAP sekä sarjaporttiprofiili SPP. Kummankin toiminnallisuus on toteutettu takaisinkutsufunktioina. Takaisinkutsufunktiot ovat toiselle funktiolle parametreina annettavia funktioita, joita pystytään kutsumaan toisen funktion kautta, kun tälle annetaan sitä vastaava kehote. Tässä esimerkissä yleisen pääsyprofiilin ja sarjaporttiprofiilin funktiot tallennetaan Bluetooth-ohjaimelle tiedossa olevaan listaan `btc_profile_cb_tab[profile_id]` vastaamaan oikeaa profiilitunnistetta funktiolla `btc_profile_cb_set(btc_pid_t profile_id, void *cb)`. Näin ollen, kun ohjaimelle tulee pyyntö tarkkailijalta johonkin näistä profiileista, se osaa kutsua kyseisen profiilin toteutusfunktiota. Profiilien funktiot ovat toteutettu käyttäen switch-case rakennetta, jonka parametrina on Bluetooth-ohjaimelta tulevat profiilin tiloja vastaavat enum-arvot. Switch-case on ehtolausetyyppi, jossa arvoa verrataan switchin sisällä oleviin case-haarojen arvoihin. Jos jokin näistä toteutuu, niin vain se haara suoritetaan [16, Luku 9].

Yleisen pääsyprofiilin toteutusfunktio `esp_bt_gap_cb()` sisältää kahden GAP:n saaman takaisinkutsun käsittelyt. Nämä ovat ilmoitus onnistuneesta autentikaatiosta sekä käsittely

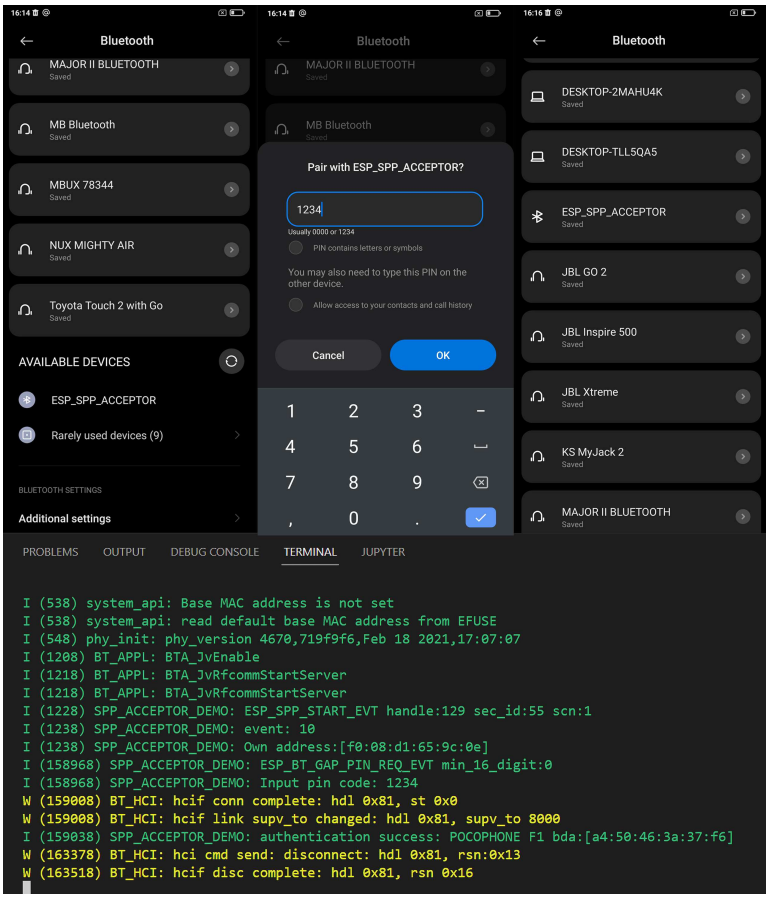
tarkkailijan lähettämään PIN-koodin kyselyyn. PIN-koodin kyselyn käsittelyssä Bluetooth-ohjaimelle välitetään viesti, jolla kerrotaan, että käskyn lähettäjä on yleinen pääsyprofiili ja käskyn tyyppi on vastaus PIN-koodikyselyyn. Tämän lisäksi välitetään argumentteina PIN-koodi, koodin pituus, kyselyn pyyntövahvistus sekä ESP32:n käyttämä Bluetooth-osoite. Ohjaimelle myös välitetään syväkopion luontifunktio.

SPP:n takaisinkutsun alustus toimii samalla tavalla, eli asetetaan `btc_profile_cb_set()`-funktiolla SPP:n toiminnasta vastaava funktio sitä vastaavaan profiilinumeroon. SPP-funktiossa on käsittelyt 12 eri tapahtumalle. Näistä oleellisimmat ovat SPP-palvelimen alustus, palvelimen käynnistys ja datan vastaanotto. SPP-palvelimen alustuksessa Bluetooth-ohjaimelle lähetetään viesti, jossa ilmoitetaan palvelimen nimi, sen rooli yhteydenmuodostuksessa, suurin tukema yhdenaikaisten linkkien määrä sekä käytössä olevan turvallisuusmaskin tyyppi. Kun SPP-palvelin käynnistetään, niin ohjaimen kautta välitetään laitteen nimi, joka näkyy esimerkiksi tarkkailijalaitteen yhdistettävien Bluetooth-laitteiden listassa. Tämän lisäksi asetetaan myös tarkkailutilat, joissa ESP näkyy tarkkailijalaitteille. Tässä ne on yhdistettävä ja yleisesti havaittava.

Loput SPP:n tilakäsittelyistä ovat yhteyden tilasta kertovia tapahtumia, kuten esimerkiksi ilmoitus yhteyden muodostuksesta, sen keskeytyksestä tai ilmoitus tarkkailijan yhteydenmuodostuspyynnöstä. Tässä esimerkissä ne vain tulostavat terminaaliin viestin siitä, että kyseinen tapahtuma on toteutunut.

4.1.3 Yhteys tarkkailijan kanssa

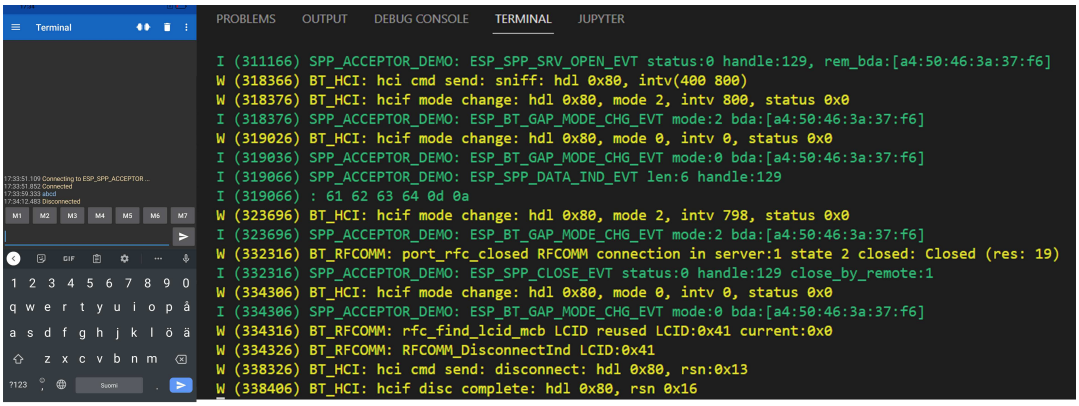
Kun koodi kirjoitetaan ESP32:lle ja ohjelma aloitetaan, se ajaa kaikki alustukset ja käynnistykset, minkä jälkeen muut laitteet pystyvät ottamaan siihen yhteyden. Tarkkailijan roolissa toimii Android-käyttöjärjestelmällä toimiva Pocophone F1-puhelin. Puhelimen näkymän sekä ESP32:n tietokoneen monitorointilokin tulosteet voi nähdä kuvasta 4.2.



Kuva 4.2. *Bluetooth-yhteyden muodostamisvaiheet*

Kun puhelimelta yritetään luoda yhteys ESP32:een, näytölle tulee pyyntö PIN-koodin syöttämisestä. Tässä vaiheessa siis ESP32 sai puhelimelta `ESP_BT_GAP_PIN_REQ_EVT`-pyynnön, johon ESP32 vastaa lähettämällä `BTC_GAP_BT_ACT_PIN_REPLY`-tyyppisen viestin Bluetooth-ohjaimen kautta puhelimelle. Kun puhelimella syötetään oikea PIN-koodi, yhteydenmuodostamisprosessi saatetaan onnistuneesti loppuun ja monitorointilokiin tulostuu puhelimen bda-osoite. Bda-osoite eli Bluetooth-laiteosoite (engl. Bluetooth Device Address) on valmistajan laitteelle määrittämä ainutlaatuinen osoite, jonka avulla laitteet pystytään erottamaan toisistaan [12, s. 155–156]. Tämän jälkeen yhteys katkaistaan, mistä kertovat isäntäohjainrajapinnan monitorointilokin tulosteet. Nyt ESP32 myös näkyy puhelimen yhdistettyjen laitteiden listassa 4.2.

Sarjaliikenteen toteutus tapahtuu käyttämällä puhelimelle asennetun Serial Bluetooth Terminal-sovelluksen avulla. Laiteparin muodostuksen jälkeen ESP32 näkyy myös sovelluksen laitelistassa. Kun siihen muodostetaan yhteys, monitorointilokiin tulostuu `ESP_SPP_SRV_OPEN_EVT`, mikä tarkoittaa sitä, että yhteys puhelimeen on avattu. Jos nyt kirjoittaa jotain sovelluksen terminaaliin, niin kyseinen teksti tulostuu monitorointilokiin siinä olevien merkkien ASCII-arvoina. Puhelimen lähettämän viestin sekä monitorointilokin tulosteet tässä vaiheessa voi nähdä kuvasta 4.3.



Kuva 4.3. *Muodostettu SPP-yhteys ja tiedon välitys*

Viestin lähetyksen jälkeen puhelimen toimesta lähetetään yhteydenkatkaisupyyntö, jolloin RFCOMM-yhteys lopetetaan ja isäntäohjainrajapinta katkaisee yhteyden.

4.2 BLE-lisälaite

BLE-toiminnallisuutta tarkastellaan käyttämällä toista ESP-IDF:ssä tarjottavaa BLE-lisälaite-esimerkkiä [2]. Esimerkin koodi on liitteessä C ja muut oleellisimmat toimintaan liittyvät funktiototeutukset liitteessä D. Tässä esimerkissä ESP32 toimii lisälaitteena BLE-yhteydessä ja mainostaa itseään sekä tarjoamiaan palveluja. Tämän lisäksi kyseisen esimerkin avulla pystyy tarkastelemaan BLE:n tukemia turvallisuusominaisuuksia, kuten väliintulohyökkäyksen estoa sekä turvallisen linkin muodostusta. Kyseiset ominaisuudet voi asettaa päälle tai pois menuconfig-terminaalikomennon kautta, mutta laitteiston rajoitteiden vuoksi väliintulohyökkäyksen estoa ei voi asettaa pelkällä mikrokontrollerilla. Väliintulohyökkäyksessä laitteiden linkin väliin tulee hyökkääjälaite, joka muodostaa yhteyden kumpaankin laitteeseen huomaamattomasti ja kaappaa niiden välillä liikkuvan datan. Väliintulohyökkäyksen esto on toteutettu Bluetoothissa 16-merkkisen koodin avulla, jota pyydetään syöttämään kummallakin laitteella. Tämä kuitenkin vaatii, että kummallakin laitteella olisi jokin keino syöttää kyseinen koodi. [12, s. 73–74]

4.2.1 Pinojärjestäjän alustaminen

Toteutus alkaa NimBLE-pinojärjestäjän käynnistämisestä funktiolla `esp_nimble_hci_and_controller_init()`. Funktio alustaa ja käynnistää Bluetooth-kontrollerin, minkä jälkeen käynnistetään isäntäohjainrajapinta. Seuraava funktio on `nimble_port_init()`. Sen avulla alustetaan isäntäpino (engl. host stack), joka suorittaa kaikki sovellustasolta tulevat tehtävät ja välittää ne Bluetooth-kontrollerille. Isäntäpinolle asetetaan myös takaisinkutsufunktioita sen konfiguraatioon. Näitä ovat pinon uudelleenkäynnistys virheen tapauksessa, isännän ja kontrollerin onnistuneen synkronoinnin jälkeinen toiminnallisuus, uusien yleisten ominaisuusprofiilien rekisteröiminen ja muistin täyttymisen hallintafunktio.

Laitteen tarjoamien palveluiden alustaminen tehdään funktiolla `gatt_svr_init()`. Se alustaa yleisen pääsyprofiilin ja ominaisuusprofiilin asettamalla näihin palvelut, joita ESP32 tulee tarjoamaan muille laitteille. Uusien palveluiden asettaminen tapahtuu funktioilla `ble_gatts_count_cfg(gatt_svr_svcs)` ja `ble_gatts_add_svcs(gatt_svr_svcs)`. Kumpikin funktio ottaa parametrina rakenteen palvelusta, jonka voi nähdä koodiesimerkistä 4.1. Ensimmäinen näistä funktioista tarkistaa, että lisättävät palvelut sisältävät vaadittavan määrän ominaisuuksia (engl. characteristics), määreitä (engl. attributes) ja määrittelijöitä (engl. descriptor). Niiden lukumäärä lisätään isäntäpinon konfiguraatioon. Toisella funktiolla itse palvelut lisätään konfiguraatioon.

Kun kaikki palvelut ja takaisinkutsufunktiot pinojärjestäjälle on asetettu, pinonjärjestäjä käynnistetään funktiolla `nimble_port_freertos_init(bleprph_host_task)`. Käynnistyksen jälkeen edellä asetettu takaisinkutsufunktio `gatt_svr_register_cb()` asettaa konfiguraatiossa määritetyt palvelut laitteelle ja `bleprph_on_sync()` kutsuu mainostamisen aloitusfunktiota `bleprph_advertise()`. Mainostuksen aloitusfunktiossa asetetaan laitteen näkyvyys yleiseksi sekä kerrotaan, että laite ei tue BR/EDR-Bluetoothia. Funktiossa myös asetetaan laitteen lähetysteho sekä nimi, joka tässä esimerkissä on nimble-bleprph. Kun mainostuksen alustaminen on tehty, kutsutaan funktiota `ble_gap_adv_start()`, joka ottaa yhdeksi parametrikseen takaisinkutsufunktion `bleprph_gap_event()`. Tässä funktiossa käsitellään kaikki yleiselle pääsyprofiilille tulevat tehtävät, kuten yhteyden muodostaminen, mainostamisen lopetus, yhteyden katkaiseminen, salasanan pyytäminen vastaanottajalaitteelta, salauksen hoitaminen sekä yhteyden päivittäminen, jos yhteydenmuodostusparametrit vaihtuvat.

4.2.2 BLE-palvelut

Kaikki koodissa alustettavat palvelut noudattavat samaa rakennetta. Esimerkki kyseisestä rakenteesta on esimerkissä 4.1. Kyseinen esimerkki alustaa palvelun, jolla pystytään hakemaan ESP32:lta satunnaisesti generoitu luku sekä kirjoittamaan ja lukemaan ESP32:n muistista jokin arvo. Palvelun rakenteessa asetetaan sen tyyppi, palvelun uniikki tunniste, ominaisuuksien takaisinkutsufunktiot, ominaisuuksien tunnisteet sekä liput.

Koodiesimerkki 4.1. *GATT palvelun rakenne*

```
static const struct ble_gatt_svc_def gatt_svr_svcs[] = {
    {
        /*** Service: Security test. */
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = &gatt_svr_svc_sec_test_uuid.u,
        .characteristics = (struct ble_gatt_chr_def[])
        { {
                /*** Characteristic: Random number generator. */
                .uuid = &gatt_svr_chr_sec_test_rand_uuid.u,
```

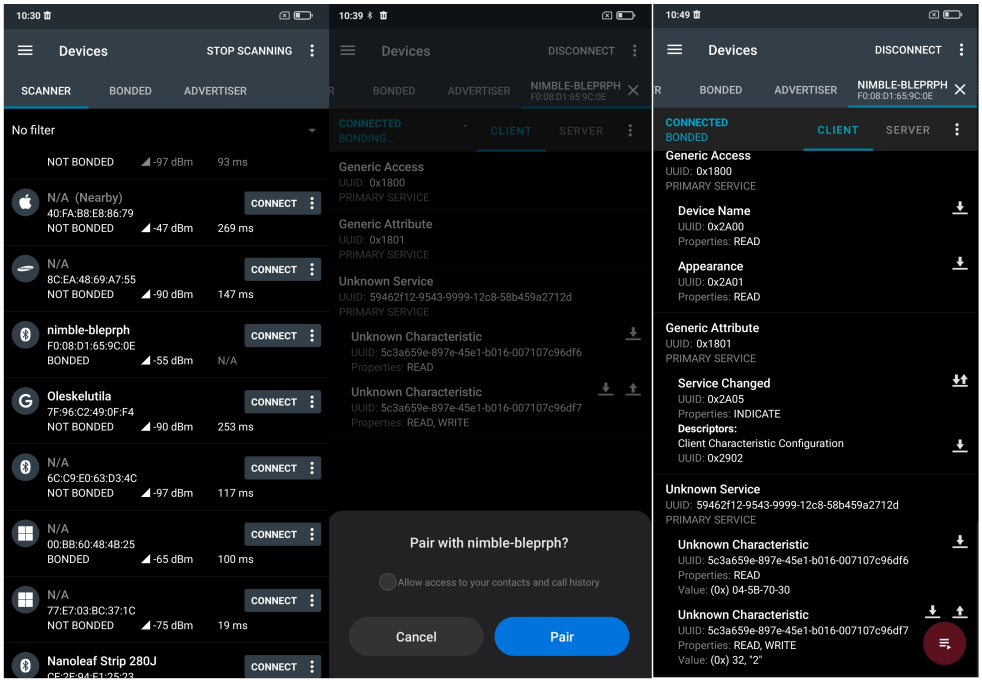
```
                .access_cb = gatt_svr_chr_access_sec_test,
                .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_READ_ENC,
            }, {
                /*** Characteristic: Static value. */
                .uuid = &gatt_svr_chr_sec_test_static_uuid.u,
                .access_cb = gatt_svr_chr_access_sec_test,
                .flags = BLE_GATT_CHR_F_READ |
                BLE_GATT_CHR_F_WRITE | BLE_GATT_CHR_F_WRITE_ENC,
            }, {
                0, /* No more characteristics in this service. */
            }
        },
    },
};
```

Tyypillä asetetaan, onko kyseinen palvelu ensisijainen vai toissijainen palvelu. Ensisijaiset palvelut näkyvät vastaanottavalle laitteelle tarjottavien palveluiden listassa ja ovat sellaisinaan sen käytettävissä. Toissijaiset palvelut ovat ensisijaisten palveluiden kutsumia ja niiden avulla toteutetaan lisäominaisuuksia ensisijaisille palveluille. Ne eivät myöskään ole näkyvissä vastaanottajalaitteelle. [12, s. 291]. Seuraavana on yleisesti uniikin tunnisteen asettaminen. Kyseiselle palvelulle ei ole ennalta määritettyä tunnistetta, joten se on generoitu satunnaisesti. Ominaisuuksille asetetaan omat tunnisteet sekä asetetaan takaisinkutsufunktiot. Nämä funktiot suoritetaan, kun kyseisen palvelun ominaisuutta kutsutaan vastaanottajalaitteella. Ominaisuuksille asetetaan myös liput. Lipuilla määritetään ominaisuuden käyttäytyminen vastaanottajalaitteen näkökulmasta, kuten luku-/kirjoitusoikeudet tai ESP32:n mahdollisuus lähettää ilmoituksia kyseiseen ominaisuuteen liittyen. Ilmoitukset ovat yksi BLE:n energiasäästömekanismeista indikaation ohella, jossa laite ilmoittaa vastaanottajalle, että sillä olisi lähetettävää dataa. Näin vastaanottajan ei tarvitse itse jatkuvasti pyytää uutta dataa lähettävältä laitteelta [12, s. 296]. Indikaatio eroaa ilmoituksesta sillä, että vastaanottajalaitteen on kerrottava onnistuneesti vastaanotetusta datasta saapuneeseen viestiin. Esimerkkipalvelussa 4.1 satunnaislukugeneraattorille on asetettuna vain lukuoikeus. Staattisen arvon ominaisuudelle sen sijaan on asetettu sekä luku, että kirjoitusoikeudet. Tämän lisäksi sille on myös lisätty mahdollisuus salattuun kirjoittamiseen, jota hyödynnetään, kun laitteiden välille on muodostettu salattu linkki.

Edellä mainitun palvelun lisäksi asetetaan myös pakolliset yleisen pääsyprofiilin ja yleisen ominaisuusprofiilin palvelut, jotka voi nähdä kuvan 4.4 kolmannesta kuvakaappauksesta. Yleisen pääsyprofiilin tarjoamat palvelut ovat laitteen nimi sekä sen tyyppi. Tyyppi on ennalta määritetty arvo, jonka avulla vastaanottajalaite voi tunnistaa BLE-laitteen esimerkiksi kuulokkeiksi tai puhelimeksi [12, s. 338]. Yleisellä ominaisuusprofiililla on vain yksi palvelu, jonka avulla pystyy muuttamaan, miten lisälaite lähettää ilmoituksia ja indikaatioita vastaanottajalle.

4.2.3 Yhteys vastaanottajaan

Kun ESP32:n BLE-toiminnallisuus on alustettu onnistuneesti ja binäärit on siirretty laitteelle, pystytään muodostamaan yhteys vastaanottajalaitteeseen. Vastaanottajana toimii edelleen Pocophone F1-puhelin, johon on asennettuna Nordic Semiconductorin tekemä nRF Connect-sovellus. Kuvasta 4.4 näkee BLE-yhteyden eri vaiheet.



Kuva 4.4. *BLE-yhteyden muodostusvaiheet*

Sovelluksen Scanner-välilehdellä pystyy näkemään kaikki lähistöllä olevat Bluetooth-laitteet, mukaan lukien nimble-bleprph. Tämän voi nähdä kuvan 4.4 ensimmäisestä kuvakaappauksesta. Yhdistämispainiketta painaessa puhelimen näytölle ilmestyy ESP32:n tarjoamat palvelut, joita olivat yleinen pääsyprofiili, yleinen ominaisuusprofiili sekä esimerkkikoodissa 4.1 asetettu palvelu. Kyseinen näkymä on kuvan 4.4 toisessa kuvakaappauksessa taustalla. Tässä tilassa on muodostettu yhteys laitteeseen ja pystytään tarkkailemaan sen tarjoamia palveluita. Monitorointilokiin tulee tuloste siitä, että yhteys laitteeseen on muodostettu.

Jotta palveluita voitaisiin käyttää, ne täytyy kuitenkin tilata. Se onnistuu esimerkiksi koittamalla lukea satunnaislukugeneraattorin arvo. Kun puhelimelta tulee siihen pyyntö, ESP32 lähettää tilausvahvistuksen laitteelle, joka täytyy hyväksyä. Hyväksymänäkymä on kuvan 4.4 toisessa kuvakaappauksessa. Hyväksymisen jälkeen puhelimelta pystyy lukemaan ESP32:n lähettämiä satunnaislukuja sekä kirjoittamaan ja lukemaan staattisen luvun arvo. Näkymää vastaanotetusta arvosta voi nähdä kuvan 4.4 kolmannessa kuvakaappauksessa. Samaan aikaan monitorointilokiin tulee tuloste siitä, että vastaanottajalaite on onnistuneesti tilannut tarjottavan palvelun.

5. YHTEENVETO

Työn tarkoituksena oli selvittää, kuinka Bluetooth- ja Bluetooth Low Energy-teknologiat ovat toteutettu. Tämä saavutettiin käymällä läpi kummankin teknologian protokollapinot ja selvittämällä oleellisimpien protokollien tehtävät siinä. Näiden lisäksi selvitettiin, että Bluetoothin korkeimmilla tasoilla toiminnallisuus on toteutettu käyttämällä profiileja, eli ennalta määritettyjä toiminnallisuusmalleja, joiden avulla taataan eri valmistajien laitteiden välinen yhteistoiminta.

Käytännön toteutusta tutkiessa selvisi, että ESP-IDF:ssä ei pysty konfiguroimaan koko Bluetooth-pinoa itse, vaan esimerkiksi fyysisen tason protokollien toteutus on tehty jo valmistajan puolesta ja se annetaan vain mikrokontrollerille luettavissa olevassa binäärimuodossa. Tämän takia Bluetoothin toiminta toteutetaan käyttämällä Bluetooth-pinonrakentajia. Pinonrakentajien avulla saadaan alustettua Bluetooth-pino ja kommunikoitua fyysisen tason protokollien kanssa ilman, että tarvitsee itse ohjelmoida sen toteutusta.

Käytännön osuudessa huomattiin myös, että laitteen peruskonfigurointi on tehty erittäin helpoksi konfiguraatiotyökalun avulla. Ohjelmoijan täytyy vain asettaa oikeat parametrit konfiguraatiotiedostoon, jolloin ESP-IDF:n kanssa tulevassa koodissa olevien ehtojen avulla se alustaa tarvittavan toiminnallisuuden. Myös Espressifin tarjoamat mallikoodit ovat hyvä pohja omien projektien aloittamiseen, sillä niiden päälle on helppo toteuttaa oma laitteistototeutus ja halutessaan muokata tarvittavia Bluetoothiin liittyviä asetuksia.

Vaikka työssä yritettiinkin selvittää Bluetoothin toiminta mahdollisimman hyvin, kyseisen työn aikana selvisi, että BR/EDR ja BLE-teknologiat ovat hyvin laajoja ja niihin kuuluu työssä mainittujen protokollien ja profiilien lisäksi paljon muutakin toiminnallisuutta. Tässä työssä ei esimerkiksi käsitelty audion välitystä Bluetoothin kautta, vaikka se onkin kuluttajamarkkinoilla yksi yleisimmistä käyttökohteista Bluetoothille. Myös se, että työssä tutkittu versio ei ollut kirjoitushetkellä uusin 5.3 tuo mahdollisuuden jatkotutkimukselle esimerkiksi sen tuomista uudistuksista verrattuna vanhempiin Bluetoothin versioihin.

LÄHTEET

[1] Andrea Aza et al. ”Bluetooth 5 performance analysis for inter-vehicular communications”. *Wireless networks* 28 (2022). ISSN: 1022-0038.

[2] *BLE Peripheral Example*. URL: https://github.com/espressif/esp-idf/tree/master/examples/bluetooth/nimble/bleprph (viitattu 12. 11. 2022).

[3] *Bluetooth.com About Us - section*. URL: https://www.bluetooth.com/about-us/ (viitattu 24. 09. 2022).

[4] *Core Specification 4.2 – Bluetooth® Technology Website*. URL: https://www.bluetooth.com/specifications/specs/core-specification-4-2/ (viitattu 24. 09. 2022).

[5] *ESP-IDF BT-SPP-ACCEPTOR demo*. URL: https://github.com/espressif/esp-idf/tree/02605f1a31be4233b97e83a0460bb458f277dfea/examples/bluetooth/bluedroid/classic_bt/bt_spp_acceptor (viitattu 22. 10. 2022).

[6] *ESP-IDF Documentation*. URL: https://www.espressif.com/en/products/sdks/esp-idf (viitattu 14. 09. 2022).

[7] *ESP-IDF Documentation: Bluetooth API*. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/bluetooth/index.html# (viitattu 14. 09. 2022).

[8] *ESP32 Bluetooth Architecture*. URL: https://www.espressif.com/sites/default/files/documentation/esp32_bluetooth_architecture_en.pdf (viitattu 24. 09. 2022).

[9] *ESP32 Series Datasheet*. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (viitattu 13. 09. 2022).

[10] *ESP32-WROOM-32D & ESP32-WROOM-32U Datasheet*. URL: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf (viitattu 13. 09. 2022).

[11] *FIREBEETLE BOARD-ESP32 USER MANUAL*. URL: https://www.manualslib.com/manual/1510500/Dfrobot-Firebeetle-Board-Esp32.html (viitattu 01. 10. 2022).

[12] Naresh Gupta. *Inside Bluetooth low energy*. eng. Second edition. Artech House, 2016. ISBN: 1-5231-3476-3.

[13] Peter Hoddie ja Lizzie Prader. *IoT Development for ESP32 and ESP8266 with JavaScript: A Practical Guide to XS and the Moddable SDK*. eng. Apress L. P, 2020. ISBN: 9781484250693.

[14] *JavaScript language considerations on embedded devices using the XS engine*. URL: https://github.com/Moddable-OpenSource/moddable/blob/public/documentation/xs/XS%20Differences.md (viitattu 14. 09. 2022).

[15] Brent A. Miller. *Bluetooth revealed : [the insider's guide to an open specification for global wireless communications]*. 1st edition. Prentice Hall PTR, 2001. ISBN: 0-13-090294-2.

[16] Mikael. Olsson. *C++ 14 Quick Syntax Reference Second Edition*. 2nd ed. 2015. Apress, 2015. ISBN: 1-4842-1727-6.

[17] Vedat Ozan Oner. *Developing IoT Projects with ESP32*. eng. Packt Publishing, 2021. ISBN: 1-83864-116-5.

[18] *Radiotaajuusmääräys 4 AC/2021M*. URL: https://www.finlex.fi/fi/viranomaiset/normi/480001/47642.

[19] K.V.S.S.S.S. Sairam, N. Gunasekaran ja S.R. Redd. ”Bluetooth in wireless communication”. *IEEE communications magazine* (2002), s. 90–96. ISSN: 0163-6804.

[20] ”Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN - Specific Requirements - Part 15”. *IEEE Std 802.15.1-2002* (2002), s. 1–473.

[21] Sherali Zeadally, Farhan Siddiqui ja Zubair Baig. ”25 Years of Bluetooth Technology”. *Future Internet* 11.9 (2019), s. 194. ISSN: 1999-5903.

LIITE A: BT-SPP-ACCEPTOR-PÄÄTIEDOSTO

```
/*
   This example code is in the Public Domain (or CC0 licensed, at your option
       .)

   Unless required by applicable law or agreed to in writing, this
   software is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
   CONDITIONS OF ANY KIND, either express or implied.
*/

#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include <stdio.h>
#include "nvs.h"
#include "nvs_flash.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "esp_bt.h"
#include "esp_bt_main.h"
#include "esp_gap_bt_api.h"
#include "esp_bt_device.h"
#include "esp_spp_api.h"

#include "time.h"
#include "sys/time.h"

#define SPP_TAG "SPP_ACCEPTOR_DEMO"
#define SPP_SERVER_NAME "SPP_SERVER"
#define EXAMPLE_DEVICE_NAME "ESP_SPP_ACCEPTOR"
#define SPP_SHOW_DATA 0
#define SPP_SHOW_SPEED 1
#define SPP_SHOW_MODE SPP_SHOW_DATA     /*Choose show mode: show data or speed
    */

static const esp_spp_mode_t esp_spp_mode = ESP_SPP_MODE_CB;

static struct timeval time_new, time_old;
```

```
static long data_num = 0;

static const esp_spp_sec_t sec_mask = ESP_SPP_SEC_AUTHENTICATE;
static const esp_spp_role_t role_slave = ESP_SPP_ROLE_SLAVE;

static char *bda2str(uint8_t *bda, char *str, size_t size)
{
    if (bda == NULL || str == NULL || size < 18) {
        return NULL;
    }

    uint8_t *p = bda;
    sprintf(str, "%02x:%02x:%02x:%02x:%02x:%02x",
            p[0], p[1], p[2], p[3], p[4], p[5]);
    return str;
}

static void print_speed(void)
{
    float time_old_s = time_old.tv_sec + time_old.tv_usec / 1000000.0;
    float time_new_s = time_new.tv_sec + time_new.tv_usec / 1000000.0;
    float time_interval = time_new_s - time_old_s;
    float speed = data_num * 8 / time_interval / 1000.0;
    ESP_LOGI(SPP_TAG, "speed(%fs ~ %fs): %f kbit/s" , time_old_s, time_new_s,
        speed);
    data_num = 0;
    time_old.tv_sec = time_new.tv_sec;
    time_old.tv_usec = time_new.tv_usec;
}

static void esp_spp_cb(esp_spp_cb_event_t event, esp_spp_cb_param_t *param)
{
    char bda_str[18] = {0};

    switch (event) {
    case ESP_SPP_INIT_EVT:
        if (param->init.status == ESP_SPP_SUCCESS) {
            ESP_LOGI(SPP_TAG, "ESP_SPP_INIT_EVT");
            esp_spp_start_srv(sec_mask, role_slave, 0, SPP_SERVER_NAME);
        } else {
            ESP_LOGE(SPP_TAG, "ESP_SPP_INIT_EVT status:%d", param->init.status
                );
        }
        break;
    case ESP_SPP_DISCOVERY_COMP_EVT:
        ESP_LOGI(SPP_TAG, "ESP_SPP_DISCOVERY_COMP_EVT");
        break;
    case ESP_SPP_OPEN_EVT:
```

```
            ESP_LOGI(SPP_TAG, "ESP_SPP_OPEN_EVT");
            break;
        case ESP_SPP_CLOSE_EVT:
            ESP_LOGI(SPP_TAG, "ESP_SPP_CLOSE_EVT status:%d handle:%d
                close_by_remote:%d", param->close.status,
                      param->close.handle, param->close.async);
            break;
        case ESP_SPP_START_EVT:
            if (param->start.status == ESP_SPP_SUCCESS) {
                ESP_LOGI(SPP_TAG, "ESP_SPP_START_EVT handle:%d sec_id:%d scn:%d",
                    param->start.handle, param->start.sec_id,
                          param->start.scn);
                esp_bt_dev_set_device_name(EXAMPLE_DEVICE_NAME);
                esp_bt_gap_set_scan_mode(ESP_BT_CONNECTABLE,
                    ESP_BT_GENERAL_DISCOVERABLE);
            } else {
                ESP_LOGE(SPP_TAG, "ESP_SPP_START_EVT status:%d", param->start.
                    status);
            }
            break;
        case ESP_SPP_CL_INIT_EVT:
            ESP_LOGI(SPP_TAG, "ESP_SPP_CL_INIT_EVT");
            break;
        case ESP_SPP_DATA_IND_EVT:
#if (SPP_SHOW_MODE == SPP_SHOW_DATA)
            /*
             * We only show the data in which the data length is less than 128
                here. If you want to print the data and
             * the data rate is high, it is strongly recommended to process them
                in other lower priority application task
             * rather than in this callback directly. Since the printing takes too
                much time, it may stuck the Bluetooth
             * stack and also have a effect on the throughput!
             */
            ESP_LOGI(SPP_TAG, "ESP_SPP_DATA_IND_EVT len:%d handle:%d",
                      param->data_ind.len, param->data_ind.handle);
            if (param->data_ind.len < 128) {
                esp_log_buffer_hex("", param->data_ind.data, param->data_ind.len);
            }
#else
            gettimeofday(&time_new, NULL);
            data_num += param->data_ind.len;
            if (time_new.tv_sec - time_old.tv_sec >= 3) {
                print_speed();
            }
#endif
            break;
        case ESP_SPP_CONG_EVT:
```

```
        ESP_LOGI(SPP_TAG, "ESP_SPP_CONG_EVT");
        break;
    case ESP_SPP_WRITE_EVT:
        ESP_LOGI(SPP_TAG, "ESP_SPP_WRITE_EVT");
        break;
    case ESP_SPP_SRV_OPEN_EVT:
        ESP_LOGI(SPP_TAG, "ESP_SPP_SRV_OPEN_EVT status:%d handle:%d, rem_bda
            :[%s]", param->srv_open.status,
                param->srv_open.handle, bda2str(param->srv_open.rem_bda,
                    bda_str, sizeof(bda_str)));
        gettimeofday(&time_old, NULL);
        break;
    case ESP_SPP_SRV_STOP_EVT:
        ESP_LOGI(SPP_TAG, "ESP_SPP_SRV_STOP_EVT");
        break;
    case ESP_SPP_UNINIT_EVT:
        ESP_LOGI(SPP_TAG, "ESP_SPP_UNINIT_EVT");
        break;
    default:
        break;
    }
}

void esp_bt_gap_cb(esp_bt_gap_cb_event_t event, esp_bt_gap_cb_param_t *param)
{
    char bda_str[18] = {0};

    switch (event) {
    case ESP_BT_GAP_AUTH_CMPL_EVT:{
        if (param->auth_cmpl.stat == ESP_BT_STATUS_SUCCESS) {
            ESP_LOGI(SPP_TAG, "authentication success: %s bda:[%s]", param->
                auth_cmpl.device_name,
                    bda2str(param->auth_cmpl.bda, bda_str, sizeof(bda_str)));
        } else {
            ESP_LOGE(SPP_TAG, "authentication failed, status:%d", param->
                auth_cmpl.stat);
        }
        break;
    }
    case ESP_BT_GAP_PIN_REQ_EVT:{
        ESP_LOGI(SPP_TAG, "ESP_BT_GAP_PIN_REQ_EVT min_16_digit:%d", param->
            pin_req.min_16_digit);
        if (param->pin_req.min_16_digit) {
            ESP_LOGI(SPP_TAG, "Input pin code: 0000 0000 0000 0000");
            esp_bt_pin_code_t pin_code = {0};
            esp_bt_gap_pin_reply(param->pin_req.bda, true, 16, pin_code);
        } else {
            ESP_LOGI(SPP_TAG, "Input pin code: 1234");
```

```
            esp_bt_pin_code_t pin_code;
            pin_code[0] = '1';
            pin_code[1] = '2';
            pin_code[2] = '3';
            pin_code[3] = '4';
            esp_bt_gap_pin_reply(param->pin_req.bda, true, 4, pin_code);
        }
        break;
    }

#if (CONFIG_BT_SSP_ENABLED == true)
    case ESP_BT_GAP_CFM_REQ_EVT:
        ESP_LOGI(SPP_TAG, "ESP_BT_GAP_CFM_REQ_EVT Please compare the numeric
            value: %d", param->cfm_req.num_val);
        esp_bt_gap_ssp_confirm_reply(param->cfm_req.bda, true);
        break;
    case ESP_BT_GAP_KEY_NOTIF_EVT:
        ESP_LOGI(SPP_TAG, "ESP_BT_GAP_KEY_NOTIF_EVT passkey:%d", param->
            key_notif.passkey);
        break;
    case ESP_BT_GAP_KEY_REQ_EVT:
        ESP_LOGI(SPP_TAG, "ESP_BT_GAP_KEY_REQ_EVT Please enter passkey!");
        break;
#endif

    case ESP_BT_GAP_MODE_CHG_EVT:
        ESP_LOGI(SPP_TAG, "ESP_BT_GAP_MODE_CHG_EVT mode:%d bda:[%s]", param->
            mode_chg.mode,
                bda2str(param->mode_chg.bda, bda_str, sizeof(bda_str)));
        break;

    default: {
        ESP_LOGI(SPP_TAG, "event: %d", event);
        break;
    }
    }
    return;
}

void app_main(void)
{
    char bda_str[18] = {0};
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
        ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
```

```
    ESP_ERROR_CHECK( ret );

    ESP_ERROR_CHECK(esp_bt_controller_mem_release(ESP_BT_MODE_BLE));

    esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
    if ((ret = esp_bt_controller_init(&bt_cfg)) != ESP_OK) {
        ESP_LOGE(SPP_TAG, "%s initialize controller failed: %s\n", __func__,
            esp_err_to_name(ret));
        return;
    }

    if ((ret = esp_bt_controller_enable(ESP_BT_MODE_CLASSIC_BT)) != ESP_OK) {
        ESP_LOGE(SPP_TAG, "%s enable controller failed: %s\n", __func__,
            esp_err_to_name(ret));
        return;
    }

    if ((ret = esp_bluedroid_init()) != ESP_OK) {
        ESP_LOGE(SPP_TAG, "%s initialize bluedroid failed: %s\n", __func__,
            esp_err_to_name(ret));
        return;
    }

    if ((ret = esp_bluedroid_enable()) != ESP_OK) {
        ESP_LOGE(SPP_TAG, "%s enable bluedroid failed: %s\n", __func__,
            esp_err_to_name(ret));
        return;
    }

    if ((ret = esp_bt_gap_register_callback(esp_bt_gap_cb)) != ESP_OK) {
        ESP_LOGE(SPP_TAG, "%s gap register failed: %s\n", __func__,
            esp_err_to_name(ret));
        return;
    }

    if ((ret = esp_spp_register_callback(esp_spp_cb)) != ESP_OK) {
        ESP_LOGE(SPP_TAG, "%s spp register failed: %s\n", __func__,
            esp_err_to_name(ret));
        return;
    }

    if ((ret = esp_spp_init(esp_spp_mode)) != ESP_OK) {
        ESP_LOGE(SPP_TAG, "%s spp init failed: %s\n", __func__,
            esp_err_to_name(ret));
        return;
    }

#if (CONFIG_BT_SSP_ENABLED == true)
```

```
    /* Set default parameters for Secure Simple Pairing */
    esp_bt_sp_param_t param_type = ESP_BT_SP_IOCAP_MODE;
    esp_bt_io_cap_t iocap = ESP_BT_IO_CAP_IO;
    esp_bt_gap_set_security_param(param_type, &iocap, sizeof(uint8_t));
#endif

    /*
     * Set default parameters for Legacy Pairing
     * Use variable pin, input pin code when pairing
     */
    esp_bt_pin_type_t pin_type = ESP_BT_PIN_TYPE_VARIABLE;
    esp_bt_pin_code_t pin_code;
    esp_bt_gap_set_pin(pin_type, 0, pin_code);

    ESP_LOGI(SPP_TAG, "Own address:[%s]", bda2str((uint8_t *)
        esp_bt_dev_get_address(), bda_str, sizeof(bda_str)));
}
```

LIITE B: BT-SPP-ACCEPTOR-LISÄFUNKTIOT

```
esp_err_t esp_bt_controller_init(esp_bt_controller_config_t *cfg)
{
    esp_err_t err;
    uint32_t btdm_cfg_mask = 0;

    //if all the bt available memory was already released, cannot initialize
        bluetooth controller
    if (btdm_dram_available_region[0].mode == ESP_BT_MODE_IDLE) {
        return ESP_ERR_INVALID_STATE;
    }

    osi_funcs_p = (struct osi_funcs_t *)malloc_internal_wrapper(sizeof(struct
        osi_funcs_t));
    if (osi_funcs_p == NULL) {
        return ESP_ERR_NO_MEM;
    }

    memcpy(osi_funcs_p, &osi_funcs_ro, sizeof(struct osi_funcs_t));
    if (btdm_osi_funcs_register(osi_funcs_p) != 0) {
        return ESP_ERR_INVALID_ARG;
    }

    if (btdm_controller_status != ESP_BT_CONTROLLER_STATUS_IDLE) {
        return ESP_ERR_INVALID_STATE;
    }

    if (cfg == NULL) {
        return ESP_ERR_INVALID_ARG;
    }

    if (cfg->controller_task_prio != ESP_TASK_BT_CONTROLLER_PRIO
            || cfg->controller_task_stack_size < ESP_TASK_BT_CONTROLLER_STACK)
                {
        return ESP_ERR_INVALID_ARG;
    }

    //overwrite some parameters
    cfg->bt_max_sync_conn = CONFIG_BTDM_CTRL_BR_EDR_MAX_SYNC_CONN_EFF;
```

```
    cfg->magic  = ESP_BT_CONTROLLER_CONFIG_MAGIC_VAL;

    if (((cfg->mode & ESP_BT_MODE_BLE) && (cfg->ble_max_conn <= 0 || cfg->
        ble_max_conn > BTDM_CONTROLLER_BLE_MAX_CONN_LIMIT))
            || ((cfg->mode & ESP_BT_MODE_CLASSIC_BT) && (cfg->bt_max_acl_conn
                <= 0 || cfg->bt_max_acl_conn >
                BTDM_CONTROLLER_BR_EDR_MAX_ACL_CONN_LIMIT))
            || ((cfg->mode & ESP_BT_MODE_CLASSIC_BT) && (cfg->bt_max_sync_conn
                 > BTDM_CONTROLLER_BR_EDR_MAX_SYNC_CONN_LIMIT))) {
        return ESP_ERR_INVALID_ARG;
    }

    ESP_LOGI(BTDM_LOG_TAG, "BT controller compile version [%s]",
        btdm_controller_get_compile_version());

    s_wakeup_req_sem = semphr_create_wrapper(1, 0);
    if (s_wakeup_req_sem == NULL) {
        err = ESP_ERR_NO_MEM;
        goto error;
    }

    esp_phy_pd_mem_init();

    esp_bt_power_domain_on();

    btdm_controller_mem_init();

    periph_module_enable(PERIPH_BT_MODULE);

    // set default sleep clock cycle and its fractional bits
    btdm_lpcycle_us_frac = RTC_CLK_CAL_FRACT;
    btdm_lpcycle_us = 2 << (btdm_lpcycle_us_frac);

    btdm_controller_set_sleep_mode(BTDM_MODEM_SLEEP_MODE_NONE);

    btdm_cfg_mask = btdm_config_mask_load();

    if (btdm_controller_init(btdm_cfg_mask, cfg) != 0) {
        err = ESP_ERR_NO_MEM;
        goto error;
    }

    btdm_controller_status = ESP_BT_CONTROLLER_STATUS_INITED;

    return ESP_OK;

error:
    if (s_wakeup_req_sem) {
```

```
        semphr_delete_wrapper(s_wakeup_req_sem);
        s_wakeup_req_sem = NULL;
    }
    return err;
}

esp_err_t esp_bt_controller_enable(esp_bt_mode_t mode)
{
    int ret;

    if (btdm_controller_status != ESP_BT_CONTROLLER_STATUS_INITED) {
        return ESP_ERR_INVALID_STATE;
    }

    //As the history reason, mode should be equal to the mode which set in
        esp_bt_controller_init()
    if (mode != btdm_controller_get_mode()) {
        return ESP_ERR_INVALID_ARG;
    }

    esp_phy_enable();

    if (btdm_controller_get_sleep_mode() == BTDM_MODEM_SLEEP_MODE_ORIG) {
        btdm_controller_enable_sleep(true);
    }

    // inititalize bluetooth baseband
    btdm_check_and_init_bb();

    ret = btdm_controller_enable(mode);
    if (ret != 0) {

        esp_phy_disable();
        return ESP_ERR_INVALID_STATE;
    }

    btdm_controller_status = ESP_BT_CONTROLLER_STATUS_ENABLED;
    ret = esp_register_shutdown_handler(bt_shutdown);
    if (ret != ESP_OK) {
        ESP_LOGW(BTDM_LOG_TAG, "Register shutdown handler failed, ret = 0x%x",
            ret);
    }

    return ESP_OK;
}

esp_err_t esp_bluedroid_init(void)
{
```

```
    btc_msg_t msg;
    future_t **future_p;
    bt_status_t ret;

    if (esp_bt_controller_get_status() != ESP_BT_CONTROLLER_STATUS_ENABLED) {
        LOG_ERROR("Controller not initialised\n");
        return ESP_ERR_INVALID_STATE;
    }

    if (bd_already_init) {
        LOG_ERROR("Bluedroid already initialised\n");
        return ESP_ERR_INVALID_STATE;
    }

    /*
    * BTC Init
    */
    ret = btc_init();
    if (ret != BT_STATUS_SUCCESS) {
        LOG_ERROR("Bluedroid Initialize Fail");
        return ESP_FAIL;
    }

    future_p = btc_main_get_future_p(BTC_MAIN_INIT_FUTURE);
    *future_p = future_new();
    if (*future_p == NULL) {
        LOG_ERROR("Bluedroid Initialize Fail!");
        return ESP_ERR_NO_MEM;
    }

    msg.sig = BTC_SIG_API_CALL;
    msg.pid = BTC_PID_MAIN_INIT;
    msg.act = BTC_MAIN_ACT_INIT;

    if (btc_transfer_context(&msg, NULL, 0, NULL) != BT_STATUS_SUCCESS) {
        LOG_ERROR("Bluedroid Initialize Fail");
        return ESP_FAIL;
    }

    if (future_await(*future_p) == FUTURE_FAIL) {
        LOG_ERROR("Bluedroid Initialize Fail");
        return ESP_FAIL;
    }

    bd_already_init = true;

    return ESP_OK;
}
```

```
esp_err_t esp_bluedroid_enable(void)
{
    btc_msg_t msg;
    future_t **future_p;

    if (!bd_already_init) {
        LOG_ERROR("Bludroid not initialised\n");
        return ESP_ERR_INVALID_STATE;
    }

    if (bd_already_enable) {
        LOG_ERROR("Bluedroid already enabled\n");
        return ESP_ERR_INVALID_STATE;
    }

    future_p = btc_main_get_future_p(BTC_MAIN_ENABLE_FUTURE);
    *future_p = future_new();
    if (*future_p == NULL) {
        LOG_ERROR("Bluedroid enable failed\n");
        return ESP_ERR_NO_MEM;
    }

    msg.sig = BTC_SIG_API_CALL;
    msg.pid = BTC_PID_MAIN_INIT;
    msg.act = BTC_MAIN_ACT_ENABLE;

    if (btc_transfer_context(&msg, NULL, 0, NULL) != BT_STATUS_SUCCESS) {
        LOG_ERROR("Bluedroid enable failed\n");
        return ESP_FAIL;
    }

    if (future_await(*future_p) == FUTURE_FAIL) {
        LOG_ERROR("Bluedroid enable failed\n");
        return ESP_FAIL;
    }

    bd_already_enable = true;

    return ESP_OK;
}

int btc_profile_cb_set(btc_pid_t profile_id, void *cb)
{
    if (profile_id < 0 || profile_id >= BTC_PID_NUM) {
        return -1;
    }
```

```
    btc_profile_cb_tab[profile_id] = cb;

    return 0;
}
```

LIITE C: BLE PERIPHERAL-PÄÄTIEDOSTO

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *  http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed under the License is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.  See the License for the
 * specific language governing permissions and limitations
 * under the License.
 */

#include "esp_log.h"
#include "nvs_flash.h"
/* BLE */
#include "esp_nimble_hci.h"
#include "nimble/nimble_port.h"
#include "nimble/nimble_port_freertos.h"
#include "host/ble_hs.h"
#include "host/util/util.h"
#include "console/console.h"
#include "services/gap/ble_svc_gap.h"
#include "bleprph.h"

static const char *tag = "NimBLE_BLE_PRPH";
static int bleprph_gap_event(struct ble_gap_event *event, void *arg);
static uint8_t own_addr_type;

void ble_store_config_init(void);

/**
```

```
 * Logs information about a connection to the console.
 */
static void
bleprph_print_conn_desc(struct ble_gap_conn_desc *desc)
{
    MODLOG_DFLT(INFO, "handle=%d our_ota_addr_type=%d our_ota_addr=",
                desc->conn_handle, desc->our_ota_addr.type);
    print_addr(desc->our_ota_addr.val);
    MODLOG_DFLT(INFO, " our_id_addr_type=%d our_id_addr=",
                desc->our_id_addr.type);
    print_addr(desc->our_id_addr.val);
    MODLOG_DFLT(INFO, " peer_ota_addr_type=%d peer_ota_addr=",
                desc->peer_ota_addr.type);
    print_addr(desc->peer_ota_addr.val);
    MODLOG_DFLT(INFO, " peer_id_addr_type=%d peer_id_addr=",
                desc->peer_id_addr.type);
    print_addr(desc->peer_id_addr.val);
    MODLOG_DFLT(INFO, " conn_itvl=%d conn_latency=%d supervision_timeout=%d "
                "encrypted=%d authenticated=%d bonded=%d\n",
                desc->conn_itvl, desc->conn_latency,
                desc->supervision_timeout,
                desc->sec_state.encrypted,
                desc->sec_state.authenticated,
                desc->sec_state.bonded);
}

/**
 * Enables advertising with the following parameters:
 *     o General discoverable mode.
 *     o Undirected connectable mode.
 */
static void
bleprph_advertise(void)
{
    struct ble_gap_adv_params adv_params;
    struct ble_hs_adv_fields fields;
    const char *name;
    int rc;

    /**
     *  Set the advertisement data included in our advertisements:
     *     o Flags (indicates advertisement type and other general info).
     *     o Advertising tx power.
     *     o Device name.
     *     o 16-bit service UUIDs (alert notifications).
     */

    memset(&fields, 0, sizeof fields);
```

```
    /* Advertise two flags:
     *     o Discoverability in forthcoming advertisement (general)
     *     o BLE-only (BR/EDR unsupported).
     */
    fields.flags = BLE_HS_ADV_F_DISC_GEN |
                   BLE_HS_ADV_F_BREDR_UNSUP;

    /* Indicate that the TX power level field should be included; have the
     * stack fill this value automatically.  This is done by assigning the
     * special value BLE_HS_ADV_TX_PWR_LVL_AUTO.
     */
    fields.tx_pwr_lvl_is_present = 1;
    fields.tx_pwr_lvl = BLE_HS_ADV_TX_PWR_LVL_AUTO;

    name = ble_svc_gap_device_name();
    fields.name = (uint8_t *)name;
    fields.name_len = strlen(name);
    fields.name_is_complete = 1;

    fields.uuids16 = (ble_uuid16_t[]) {
        BLE_UUID16_INIT(GATT_SVR_SVC_ALERT_UUID)
    };
    fields.num_uuids16 = 1;
    fields.uuids16_is_complete = 1;

    rc = ble_gap_adv_set_fields(&fields);
    if (rc != 0) {
        MODLOG_DFLT(ERROR, "error setting advertisement data; rc=%d\n", rc);
        return;
    }

    /* Begin advertising. */
    memset(&adv_params, 0, sizeof adv_params);
    adv_params.conn_mode = BLE_GAP_CONN_MODE_UND;
    adv_params.disc_mode = BLE_GAP_DISC_MODE_GEN;
    rc = ble_gap_adv_start(own_addr_type, NULL, BLE_HS_FOREVER,
                           &adv_params, bleprph_gap_event, NULL);
    if (rc != 0) {
        MODLOG_DFLT(ERROR, "error enabling advertisement; rc=%d\n", rc);
        return;
    }
}

/**
 * The nimble host executes this callback when a GAP event occurs.  The
 * application associates a GAP event callback with each connection that forms
    .
```

```
 * bleprph uses the same callback for all connections.
 *
 * param event The type of event being signalled.*
    param ctxt                  Various information pertaining to the event.
 * param arg Application-specified argument; unused by* bleprph.**
    return                      0 if the application successfully handled the
 *                                 event; nonzero on failure.  The semantics
 *                                 of the return code is specific to the
 *                                 particular GAP event being signalled.
 */
static int
bleprph_gap_event(struct ble_gap_event *event, void *arg)
{
    struct ble_gap_conn_desc desc;
    int rc;

    switch (event->type) {
    case BLE_GAP_EVENT_CONNECT:
        /* A new connection was established or a connection attempt failed. */
        MODLOG_DFLT(INFO, "connection %s; status=%d ",
                    event->connect.status == 0 ? "established" : "failed",
                    event->connect.status);
        if (event->connect.status == 0) {
            rc = ble_gap_conn_find(event->connect.conn_handle, &desc);
            assert(rc == 0);
            bleprph_print_conn_desc(&desc);
        }
        MODLOG_DFLT(INFO, "\n");

        if (event->connect.status != 0) {
            /* Connection failed; resume advertising. */
            bleprph_advertise();
        }
        return 0;

    case BLE_GAP_EVENT_DISCONNECT:
        MODLOG_DFLT(INFO, "disconnect; reason=%d ", event->disconnect.reason);
        bleprph_print_conn_desc(&event->disconnect.conn);
        MODLOG_DFLT(INFO, "\n");

        /* Connection terminated; resume advertising. */
        bleprph_advertise();
        return 0;

    case BLE_GAP_EVENT_CONN_UPDATE:
        /* The central has updated the connection parameters. */
        MODLOG_DFLT(INFO, "connection updated; status=%d ",
                    event->conn_update.status);
```

```
        rc = ble_gap_conn_find(event->conn_update.conn_handle, &desc);
        assert(rc == 0);
        bleprph_print_conn_desc(&desc);
        MODLOG_DFLT(INFO, "\n");
        return 0;

    case BLE_GAP_EVENT_ADV_COMPLETE:
        MODLOG_DFLT(INFO, "advertise complete; reason=%d",
                    event->adv_complete.reason);
        bleprph_advertise();
        return 0;

    case BLE_GAP_EVENT_ENC_CHANGE:
        /* Encryption has been enabled or disabled for this connection. */
        MODLOG_DFLT(INFO, "encryption change event; status=%d ",
                    event->enc_change.status);
        rc = ble_gap_conn_find(event->enc_change.conn_handle, &desc);
        assert(rc == 0);
        bleprph_print_conn_desc(&desc);
        MODLOG_DFLT(INFO, "\n");
        return 0;

    case BLE_GAP_EVENT_SUBSCRIBE:
        MODLOG_DFLT(INFO, "subscribe event; conn_handle=%d attr_handle=%d "
                    "reason=%d prevn=%d curn=%d previ=%d curi=%d\n",
                    event->subscribe.conn_handle,
                    event->subscribe.attr_handle,
                    event->subscribe.reason,
                    event->subscribe.prev_notify,
                    event->subscribe.cur_notify,
                    event->subscribe.prev_indicate,
                    event->subscribe.cur_indicate);
        return 0;

    case BLE_GAP_EVENT_MTU:
        MODLOG_DFLT(INFO, "mtu update event; conn_handle=%d cid=%d mtu=%d\n",
                    event->mtu.conn_handle,
                    event->mtu.channel_id,
                    event->mtu.value);
        return 0;

    case BLE_GAP_EVENT_REPEAT_PAIRING:
        /* We already have a bond with the peer, but it is attempting to
         * establish a new secure link.  This app sacrifices security for
         * convenience: just throw away the old bond and accept the new link.
         */

        /* Delete the old bond. */
```

```
        rc = ble_gap_conn_find(event->repeat_pairing.conn_handle, &desc);
        assert(rc == 0);
        ble_store_util_delete_peer(&desc.peer_id_addr);

        /* Return BLE_GAP_REPEAT_PAIRING_RETRY to indicate that the host
            should
         * continue with the pairing operation.
         */
        return BLE_GAP_REPEAT_PAIRING_RETRY;

    case BLE_GAP_EVENT_PASSKEY_ACTION:
        ESP_LOGI(tag, "PASSKEY_ACTION_EVENT started \n");
        struct ble_sm_io pkey = {0};
        int key = 0;

        if (event->passkey.params.action == BLE_SM_IOACT_DISP) {
            pkey.action = event->passkey.params.action;
            pkey.passkey = 123456; // This is the passkey to be entered on
                peer
            ESP_LOGI(tag, "Enter passkey %d on the peer side", pkey.passkey);
            rc = ble_sm_inject_io(event->passkey.conn_handle, &pkey);
            ESP_LOGI(tag, "ble_sm_inject_io result: %d\n", rc);
        } else if (event->passkey.params.action == BLE_SM_IOACT_NUMCMP) {
            ESP_LOGI(tag, "Passkey on device's display: %d", event->passkey.
                params.numcmp);
            ESP_LOGI(tag, "Accept or reject the passkey through console in
                this format -> key Y or key N");
            pkey.action = event->passkey.params.action;
            if (scli_receive_key(&key)) {
                pkey.numcmp_accept = key;
            } else {
                pkey.numcmp_accept = 0;
                ESP_LOGE(tag, "Timeout! Rejecting the key");
            }
            rc = ble_sm_inject_io(event->passkey.conn_handle, &pkey);
            ESP_LOGI(tag, "ble_sm_inject_io result: %d\n", rc);
        } else if (event->passkey.params.action == BLE_SM_IOACT_OOB) {
            static uint8_t tem_oob[16] = {0};
            pkey.action = event->passkey.params.action;
            for (int i = 0; i < 16; i++) {
                pkey.oob[i] = tem_oob[i];
            }
            rc = ble_sm_inject_io(event->passkey.conn_handle, &pkey);
            ESP_LOGI(tag, "ble_sm_inject_io result: %d\n", rc);
        } else if (event->passkey.params.action == BLE_SM_IOACT_INPUT) {
            ESP_LOGI(tag, "Enter the passkey through console in this format->
                key 123456");
            pkey.action = event->passkey.params.action;
```

```
            if (scli_receive_key(&key)) {
                pkey.passkey = key;
            } else {
                pkey.passkey = 0;
                ESP_LOGE(tag, "Timeout! Passing 0 as the key");
            }
            rc = ble_sm_inject_io(event->passkey.conn_handle, &pkey);
            ESP_LOGI(tag, "ble_sm_inject_io result: %d\n", rc);
        }
        return 0;
    }

    return 0;
}

static void
bleprph_on_reset(int reason)
{
    MODLOG_DFLT(ERROR, "Resetting state; reason=%d\n", reason);
}

static void
bleprph_on_sync(void)
{
    int rc;

    rc = ble_hs_util_ensure_addr(0);
    assert(rc == 0);

    /* Figure out address to use while advertising (no privacy for now) */
    rc = ble_hs_id_infer_auto(0, &own_addr_type);
    if (rc != 0) {
        MODLOG_DFLT(ERROR, "error determining address type; rc=%d\n", rc);
        return;
    }

    /* Printing ADDR */
    uint8_t addr_val[6] = {0};
    rc = ble_hs_id_copy_addr(own_addr_type, addr_val, NULL);

    MODLOG_DFLT(INFO, "Device Address: ");
    print_addr(addr_val);
    MODLOG_DFLT(INFO, "\n");
    /* Begin advertising. */
    bleprph_advertise();
}

void bleprph_host_task(void *param)
```

```
{
    ESP_LOGI(tag, "BLE Host Task Started");
    /* This function will return only when nimble_port_stop() is executed */
    nimble_port_run();

    nimble_port_freertos_deinit();
}

void
app_main(void)
{
    int rc;

    /* Initialize NVS âĂŤ it is used to store PHY calibration data */
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
        ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK(ret);

    ESP_ERROR_CHECK(esp_nimble_hci_and_controller_init());

    nimble_port_init();
    /* Initialize the NimBLE host configuration. */
    ble_hs_cfg.reset_cb = bleprph_on_reset;
    ble_hs_cfg.sync_cb = bleprph_on_sync;
    ble_hs_cfg.gatts_register_cb = gatt_svr_register_cb;
    ble_hs_cfg.store_status_cb = ble_store_util_status_rr;

    ble_hs_cfg.sm_io_cap = CONFIG_EXAMPLE_IO_TYPE;

    ble_hs_cfg.sm_sc = 1;


    rc = gatt_svr_init();
    assert(rc == 0);

    /* Set the default device name. */
    rc = ble_svc_gap_device_name_set("nimble-bleprph");
    assert(rc == 0);

    /* XXX Need to have template for store */
    ble_store_config_init();

    nimble_port_freertos_init(bleprph_host_task);
```

```
    /* Initialize command line interface to accept input from user */
    rc = scli_init();
    if (rc != ESP_OK) {
        ESP_LOGE(tag, "scli_init() failed");
    }
}
```

LIITE D: BLE PERIPHERAL-LISÄFUNKTIOT

```
esp_err_t esp_nimble_hci_and_controller_init(void)
{
    esp_err_t ret;

    esp_bt_controller_mem_release(ESP_BT_MODE_CLASSIC_BT);

    esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();

    if ((ret = esp_bt_controller_init(&bt_cfg)) != ESP_OK) {
        return ret;
    }

    if ((ret = esp_bt_controller_enable(ESP_BT_MODE_BLE)) != ESP_OK) {
        return ret;
    }
    return esp_nimble_hci_init();
}

void nimble_port_init(void)
{
    void os_msys_init(void);
    void ble_store_ram_init(void);

    /* Initialize default event queue */
    ble_npl_eventq_init(&g_eventq_dflt);

    os_msys_init();

    ble_hs_init();

    /* XXX Need to have template for store */
    ble_store_ram_init();
}

int gatt_svr_init(void)
{
    int rc;
```

```
    ble_svc_gap_init();
    ble_svc_gatt_init();

    rc = ble_gatts_count_cfg(gatt_svr_svcs);
    if (rc != 0) {
        return rc;
    }

    rc = ble_gatts_add_svcs(gatt_svr_svcs);
    if (rc != 0) {
        return rc;
    }

    return 0;
}

ble_gatts_count_cfg(const struct ble_gatt_svc_def *defs)
{
    struct ble_gatt_resources res = { 0 };
    int rc;

    rc = ble_gatts_count_resources(defs, &res);
    if (rc != 0) {
        return rc;
    }

    ble_hs_max_services += res.svcs;
    ble_hs_max_attrs += res.attrs;

    /* Reserve an extra CCCD for the cache. */
    ble_hs_max_client_configs +=
        res.cccds * (MYNEWT_VAL(BLE_MAX_CONNECTIONS) + 1);

    return 0;
}

static const struct ble_gatt_svc_def ble_svc_gatt_defs[] = {
    {
        /*** Service: GATT */
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = BLE_UUID16_DECLARE(BLE_GATT_SVC_UUID16),
        .characteristics = (struct ble_gatt_chr_def[]) { {
            .uuid = BLE_UUID16_DECLARE(
                BLE_SVC_GATT_CHR_SERVICE_CHANGED_UUID16),
            .access_cb = ble_svc_gatt_access,
            .val_handle = &ble_svc_gatt_changed_val_handle,
            .flags = BLE_GATT_CHR_F_INDICATE,
        }, {
```

```
            0, /* No more characteristics in this service. */
        } },
    },

    {
        0, /* No more services. */
    },
};

int ble_gatts_add_svcs(const struct ble_gatt_svc_def *svcs)
{
    void *p;
    int rc;

    ble_hs_lock();
    if (!ble_gatts_mutable()) {
        rc = BLE_HS_EBUSY;
        goto done;
    }

    p = realloc(ble_gatts_svc_defs,
                (ble_gatts_num_svc_defs + 1) * sizeof *ble_gatts_svc_defs);
    if (p == NULL) {
        rc = BLE_HS_ENOMEM;
        goto done;
    }

    ble_gatts_svc_defs = p;
    ble_gatts_svc_defs[ble_gatts_num_svc_defs] = svcs;
    ble_gatts_num_svc_defs++;

    rc = 0;

done:
    ble_hs_unlock();
    return rc;
}

void nimble_port_freertos_init(TaskFunction_t host_task_fn)
{
    /*
     * Create task where NimBLE host will run. It is not strictly necessary to
     * have separate task for NimBLE host, but since something needs to handle
     * default queue it is just easier to make separate task which does this.
     */
    xTaskCreatePinnedToCore(host_task_fn, "ble", NIMBLE_STACK_SIZE,
                NULL, (configMAX_PRIORITIES - 4), &host_task_h, NIMBLE_CORE);
}
```

```
void gatt_svr_register_cb(struct ble_gatt_register_ctxt *ctxt, void *arg)
{
    char buf[BLE_UUID_STR_LEN];

    switch (ctxt->op) {
    case BLE_GATT_REGISTER_OP_SVC:
        MODLOG_DFLT(DEBUG, "registered service %s with handle=%d\n",
                    ble_uuid_to_str(ctxt->svc.svc_def->uuid, buf),
                    ctxt->svc.handle);
        break;

    case BLE_GATT_REGISTER_OP_CHR:
        MODLOG_DFLT(DEBUG, "registering characteristic %s with "
                    "def_handle=%d val_handle=%d\n",
                    ble_uuid_to_str(ctxt->chr.chr_def->uuid, buf),
                    ctxt->chr.def_handle,
                    ctxt->chr.val_handle);
        break;

    case BLE_GATT_REGISTER_OP_DSC:
        MODLOG_DFLT(DEBUG, "registering descriptor %s with handle=%d\n",
                    ble_uuid_to_str(ctxt->dsc.dsc_def->uuid, buf),
                    ctxt->dsc.handle);
        break;

    default:
        assert(0);
        break;
    }
}
```