Våren 2021

Introduksjon til IoT

med bruk av NB-IoT og Nordic Thingy

Audun Brabrand
IES - NTNU

NTNU

Kunnskap for en bedre verden

Utvikling av introduksjonsprosjekt som eksempel på bruk av Nordic Thingy:91 i IoT sammenheng

Audun Brabrand
Veileder: Arne Midjo
Våren 2021

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

Audun Brabrand

NTNU
Kunnskap for en bedre verden

NTNU

Kunnskap for en bedre verden

Institutt for elektroniske systemer

Oppgavens tittel: Introduksjon til IoT med NB-IoT og Nordic Thingy **Project title:** Introduction to IoT with NB-IoT and Nordic Thingy	**Dato start:** 01.03.2021 **Dato levering:** 02.08.2021 **Gradering:** Åpen for alle **Antall sider:** 39
Gruppe: Audun Brabrand (AB) Tlf: 99162069 E-post: audubrab@stud.ntnu.no	**Veileder:** Arne Midjo (AM) arne.midjo@ntnu.no
Studieretning: Elektronikk	**Oppdragsgiver:** NTNU

Sammendrag: Denne rapporten tar for seg IoT som begrep og gir en innføring i hvordan man kan utvikle egne IoT applikasjoner basert på mikrobrikker fra Nordic Semiconductor som kjører Zephyr RTOS.

Abstract: This thesis contains a report about IoT, what it is and how we can develop IoT applications with Nordic Semiconductor components and Zephyr RTOS.

Nøkkelord/Keywords: IoT, MQTT, JSON, Zephyr RTOS, Thingy:91, Thingy:52, nRF9160dk, nRF9160, nRF52840, NB-IoT, LTE-M, BLE, Bluetooth MESH

Abstract

IoT is a comprehensive subject covering many technological areas. Plenty of information is available on the Internet, but it is not easy to know where to start to get a basic overview.

Nordic Semiconductor has developed a rich set of components suitable for use in wireless IoT applications. Software development for these are based on the Zephyr Real Time Operating System. Zephyr includes a comprehensive library of drivers and protocols suitable for use in IoT applications based on Nordic's chips. Lots of documentation and examples are available, but my experience is that a substantial technological background is required to get a good understanding.

I will in this thesis present basic technologies behind IoT, with special focus on wireless sensor networks communicating over LTE, based on components from Nordic Semiconductor and the Zephyr Real Time Operating System.

Simple example programs will be explained and demonstrated, and I will show how I modified an existing demo application to include my own set of commands to demonstrate two-way LTE communication between Nordic's "nRF Cloud" and a development board. I will use development kit nRF9160dk and Thingy:91 from Nordic during software development and demonstrations.

My hope is that this thesis will give the reader a good starting point for further development of IoT applications based on Zephyr RTOS and components from Nordic Semiconductor.

Sammendrag

IoT er et veldig vidt tema med en bratt læringskurve, det er hardt å komme i gang og det kreves pågangsmot. Under forprosjektperioden innså jeg at det finnes få veiledninger til grunnleggende IoT-programmering med Zephyr RTOS og mikrobrikker fra Nordic. Jeg håper med denne rapporten å skape en god veiledning (tutorial) for folk som ønsker å sette seg inn i temaet.

Rapporten vil ta for seg IoT som begrep og vinkle det inn mot hvordan man benytter ferdige driverbiblioteker for å bygge trådløse sensorsystemer som baserer seg på LTE kommunikasjon. Systemet vil baseres på Zephyr RTOS og med mikrobrikker fra Nordic Semiconductor. Jeg vil benytte nRF9160dk og Thingy:91 under utvikling av programkode. Målet er at en leser skal få innsikt nok til å lettere kunne forstå den dokumentasjonen som Nordic selv har skrevet, og dermed kan jobbe frem mot å utvikle egne IoT produkter.

Forord

Oppgaven og problemstillingen er utarbeidet i samarbeid mellom IoT laben på NTNU, Arne Midjo, Olav Aleksander Myrvang og meg selv, Audun Brabrand. Oppgaven ble omformulert under forprosjektrapporten da vi fikk litt bedre innblikk i omfanget av et slikt prosjekt. Arne Midjo er min veileder under prosjektperioden. Nordic Semiconductor har også vært til stor hjelp med veiledning underveis i prosjektperioden med problemer og utfordringer jeg har støtt på. Takk til Anders Storrø, Erik Robstad og Irene Sollie fra Nordic. I tillegg må en takk sendes til min far, Tord Brabrand, for assistanse med JSON parser og korrekturlesning.

Innholdsfortegnelse

Abstract iii
Sammendrag iii
Forord iii
Innholdsfortegnelse iv
Figurliste v
Akronymer vi
1. Introduksjon 1
1.1 Bakgrunn 1
1.2 Problemstilling 1
1.3 Resultatmål 1
1.4 Disposisjon 2
2. Teori 3
2.1 Hva er IoT? 3
2.2 Hvem er Nordic Semiconductor? 4
2.3 Thingy serien 4
2.4 nRF9160 4
2.5 nRF52 serien 4
2.6 Programvareverktøy 5
2.7 Zephyr RTOS 5
2.8 Dokumentasjon 5
3. Kode - Grunnleggende 6
3.1 Threads 6
3.1.1 Opprette en *thread* 7
3.1.2 Workqueue 8
3.1.3 Styre LEDs 9
3.1.4 Oppsummering 10
4. Kode – Opprette forbindelse til sky 12
4.1 MQTT og JSON 13
4.2 Cloud_client 15
4.3 Kode for oppkobling 16
5. Kode – motta data fra sky 20
5.1 Sensor avlesning 22
5.2 Oppsummering 24
6. Etterarbeid 25
7. Refleksjon 26

8. Kjekt å vite 27
8.1 SPI og I2C 27
8.2 UART 27
8.3 Nettverksstandard - OSI modellen 27
9. Referanser 28

Figurliste

Figur 1 – Illustrasjon IoT nettverk 3
Figur 2 – Thingy:91 4
Figur 3 – nRF9160 SiP 4
Figur 4 – nRF52840 SoC 4
Figur 5 – nRF9160 arkitektur 6
Figur 6 – Filstruktur 6
Figur 7 – Deklarere en thread 7
Figur 8 – "Go to definition" 7
Figur 9 – Workqueue 8
Figur 10 – Blinky 9
Figur 11 – Fra terminal 10
Figur 12 – Understanding the basics 11
Figur 13 – IoT med Nordic Semiconductor 12
Figur 14 – JSON 13
Figur 15 – MQTT forklart 15
Figur 16 – Cloud message 15
Figur 17 – Cloud_Client - main 16
Figur 18 – Cloud_Client - cloud_event_handler 17
Figur 19 – CONFIG_ CLOUD _MESSAGE 19
Figur 20 – JSON mottatt 21
Figur 21 – Attributter 21
Figur 22 – nRF Cloud 22
Figur 24 – Temperatur 22
Figur 23 – TEMP_REPORTING 22
Figur 25 – Sensoravlesning 23
Figur 26 – SPI 27

Akronymer

I^2C Inter Integrated Circuit

ADC Analog to Digital Converter

API Application Programming Interface

BLE Bluetooth Low Energy

BT Bluetooth

CLK(CK) Clock

DAC Digital to Analog Converter

DK Development kit

FIFO First in-first out

GPIO General-purpose input/output

GPS Global Positioning System

HTTP HyperText Transfer Protocol

IMEI International Mobile Equipment Identity

IoT Internet of Things

IP Internet Protocol

IRQ Interrupt Request (hardware)

ISR Interrupt Service Routine (funksjon)

LPWAN Low Power Wide Area Network

LTE-M Long Term Evolution for Machines

MQTT Message Queueing Telemetry Transport

NB-IoT Narrow-Band Internet of Things

PSM Power Saving Mode

PWM Pulse Width Modulation

RAM Random Access Memory

RAT Requested Active Time

RF Radio Frequency

RTOS Real Time Operating System

RX Receive

SIM Subscriber Identification Module

SiP System-in-Package

SoC System on Chip

SPI Serial Peripheral Interface

TCP Transmission Control Protocol

TX Transmit

UART Universal Asynchronous Receiver-Transmitter

UDP User Datagram Protocol

VLAN Virtual Local Area Network

1. Introduksjon

1.1 Bakgrunn

Jeg har alltid vært fasinert av kommunikasjon over mobile nettverk, men har ikke hatt tid til å sette meg inn i emnet før under denne bacheloren. Jeg fattet derfor raskt interessen for de bacheloroppgavene som omhandlet 4G/5G kommunikasjon og IoT applikasjoner, da de ble publisert vinteren 2020. Allerede før jul startet prosessen med å sette seg inn i denne typen teknologi i Norge. Telenor var tidlig ute med å bygge nødvendig infrastruktur for å være bransjeledende innen IoT nettverk, det var derfor et logisk sted å starte. I nettbutikken til Telenor finner man Nordic Thingy:91. Thingy:91 en komplett prototypenhet med batteri, 4G (LTE) modem og en rekke sensorer, utviklet og designet av Nordic Semiconductor. (1) (2)

Det ble en litt annerledes prosjektperiode grunnet innledende oppstartstrøbbel utenfor min kontroll. Dette gjorde at jeg endte opp uten gruppe og arbeider dermed alene om å løse oppgaven. Dette gjorde også at jeg fikk en viss grad av frihet til å formulere oppgaven etter egent ønske. Uten tidligere erfaring med hverken RTOS programmering eller større software-biblioteker ble det vanskelig å avgjøre hva som kunne være realistisk størrelse på en slik oppgave. De første utkastene til oppgaver ble dermed for ambisiøse og den endelige formuleringen kom frem mot slutten av forprosjektperioden. Oppgaven er utformet i samarbeid mellom IoT laben på NTNU, Arne Midjo og meg selv. Vi fikk innspill fra Nordic Semiconductor angående Bluetooth som gjorde at hovedfokuset blir rettet mer mot kommunikasjon over LTE nettverk.

Rapporten skal gi et godt grunnlag for å forstå hva IoT er og hvordan man utvikler egne IoT applikasjoner basert på systemene til Nordic. Det fordrer at man har grunnleggende kunnskaper om C programmering samt pågangsmot for å få fullt utbytte av denne rapporten.

1.2 Problemstilling

Etter å ha justert ambisjonsnivået for oppgaven et par ganger kom vi omsider frem til en problemstilling som lyder slik:

"*Innhente og bygge kunnskap for å forstå hvordan man på en effektiv og praktiske måte kan utvikle og implementere IoT-løsninger i eksisterende og fremtidige systemer. Prosessen skal resultere i en rapport som fremtidige studenter kan ha nytte av for å raskere sette seg inn i IoT-universet og benytte chiper fra Nordic Semiconductor i sine egne løsninger.*" (3)

1.3 Resultatmål

Prosjektets resultatmål ble også justert etter ambisjonsnivået underveis, fra å bedømme funksjonaliteten til et ferdig system legger vi i stedet mer vekt på innholdet i rapporten. Fra prosjektmanualen har vi at "*Resultatmålet beskriver hva som konkret skal foreligge som resultat når prosjektet er ferdig*". (4)

Produktets kvaliteter

- Selvforklarende og pedagogisk
- Rett på sak med relevant informasjon om det tema man til enhver tid snakker om.
- Mye eksempler og forklaringer på hva eksemplene tar for seg
- Bygge opp systemet bit for bit så man skaffer seg oversikt underveis
- Gjøre systemet utvidbart med mange bruksområder
- Legge opp til videre etterarbeid

De nye resultatmålene er rettet mer mot å gjøre rapporten oversiktlig og pedagogisk. Dette skal bidra til å gi leseren motivasjon og ideer til et etterarbeid. Etter avtale med veileder Arne Midjo har jeg gått noe bort fra den klassiske oppbygningen av en bachelorrapport for nettopp å gi den flyten man trenger for å sette seg inn i stoffet.

1.4 Disposisjon

For å gjøre rapporten så lettlest som mulig får leseren presentert relevant teori i en rekkefølge som gir mening i forhold til flyten i prosjektet. Oppbygningen er basert på mine erfaringer underveis i prosjektet i et nytt utviklingsmiljø. Underveis i rapporten vil det også bli henvist til artikler og nettsider som ble ansett som gode kilder til informasjon. Det vil være nyttig og ta for seg disse underveis i gjennomlesningen for å til enhver tid ha best mulig oversikt over tema. Henvisninger i klammeparentes er ansett som nyttige dersom man vil lese mer om et tema, men de er ikke nødvendig for forståelsen av innholdet i rapporten.

Siden en viktig del av denne rapporten er å være pedagogisk vil metode og teori bli noe blandet. Det vil være gunstig å presentere teori for så å vise hvordan det skal brukes. Den vil allikevel begynne med en introduksjon og forklaring på hva IoT er og hvem Nordic Semiconductors er. Vi vil i kapittel 3 gå nærmere inn på detaljene i et typisk IoT system.

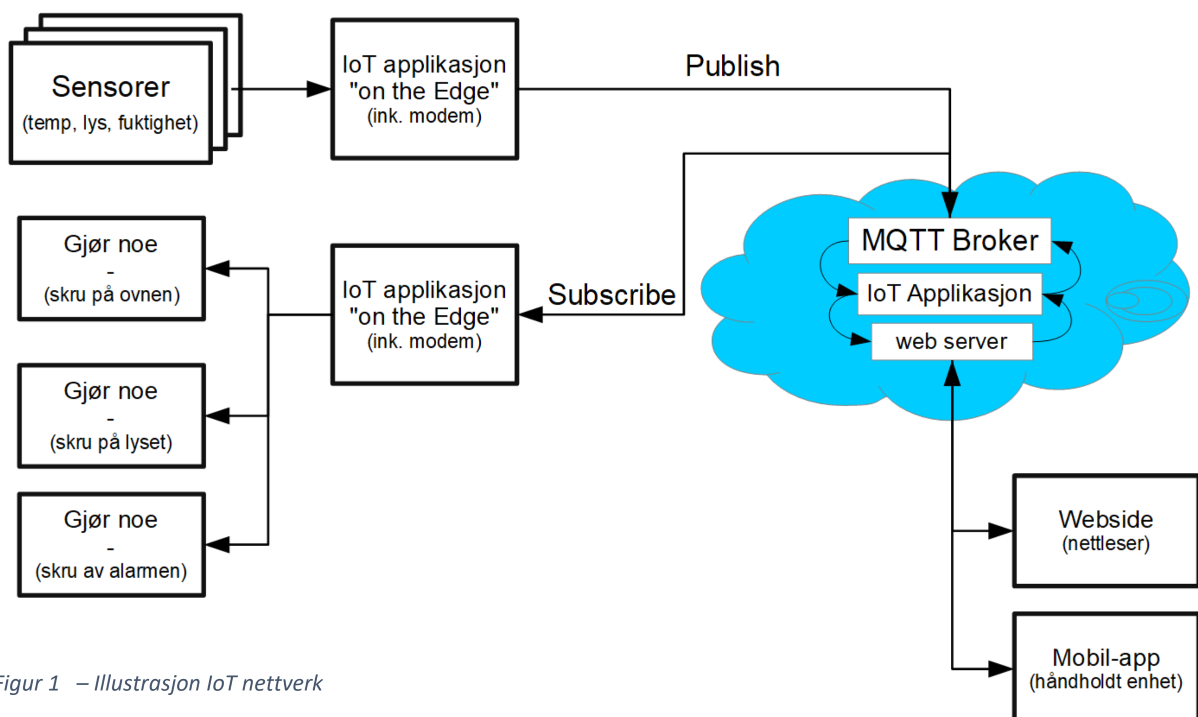
Videre vil resultater og diskusjon bli sydd noe sammen med mine erfaringer. Avslutningsvis trekker jeg frem flere temaer det vil være aktuelt å se videre på, og hva som kan være interessant å jobbe videre med som etterarbeid. Etter et slikt prosjekt vil det alltid være mange løse tråder man kan nøste videre i for å skaffe seg mer og bredere kunnskap om et slikt fagfelt.

2. Teori

2.1 Hva er IoT?

Internet of Things har kommet opp som et begrep for enheter som er koblet på internett og som samhandler uten å aktivt måtte styres av en bruker. Alle enheter har sin unike ID og kan aksesseres individuelt. Ofte vil man ha behov for å kunne koble flere enheter sammen i et lokalt nettverk for å utføre gitte oppgaver, slike enheter er typisk sensorer og aktuator. Eksempel på bruksområde kan være temperaturstyring av en hall. Her vil man da kunne ha flere temperatursensorer rundt om som kommuniserer med varmeapparatet for å opprettholde en jevn og god temperatur i lokalet. (5)

Til tross for at systemet normalt ikke har behov for interaksjon med et menneske er det ofte nødvendig første gang systemet skal konfigureres. Konfigurasjonen foregår stort sett over et web-grensesnitt via en web-server. Avhengig av design vil man også kunne følge driften av systemet i sanntid, lese av sensordata, sjekke status på aktuatoren og lese feilmeldinger. Flyten i et slikt system er vist i figur 1.



Figur 1 – Illustrasjon IoT nettverk

Til et tenkt bruksområdet har vi behov for en rekke sensorer. Disse sensorene er plassert der det er gunstig med tanke på hva de skal måle. Sensorene er innenfor Bluetooth-rekkevidde til hverandre, dette gjør det mulig å benytte Bluetooth som kommunikasjonskanal mellom dem. Sensorene snakker i sin tur med en sentral som gjør en viss grad av datahåndtering direkte, dette kallers prosessering "*on the edge*". Denne sentralen kommuniserer igjen med en web-server via mobilt internett. På webserveren har man programvare for MQTT Broker som behandler meldingsflyten i systemet, samt en IoT Applikasjonsserver som tar seg av oppsett og funksjonalitet ved systemet. En bruker kan aksessere web-serveren via en nettløsning, ofte en nettside. I dette nettverket vil det være en flyt av "*Publish*" og "*Subscribe*" meldinger som forteller hva dataene som transporteres gjelder. Vi vil snakke mer om dette i kapittel 4.

2.2 Hvem er Nordic Semiconductor?

Nordic Semiconductor, heretter kalt Nordic, ble etablert i Trondheim i 1983 av fire studenter fra NTNU. Firmaet har spesialisert seg på laveffekt mikrobrikker for trådløse applikasjoner. Brikkene er brukt i en rekke trådløse produkter, deriblant mus og tastaturer fra Logitech og Microsoft. I 2018 ble det lansert en SiP (System-in-Package) med LTE-modem og GPS mottaker rettet mot IoT markedet, og i disse dager lanserer de en ny mikrobrikke med WiFi støtte. Kombinert med de allerede eksisterende Bluetooth brikkene har de nå alt som skal til for å utvikle komplette IoT løsninger. Nordic har også utviklet et komplett driverbibliotek for mikrobrikkene sine med grensesnitt mot Zephyr RTOS. (6)

Siden oppstarten har Nordic ekspandert med over tusen ansatte og med kontorer i minst ni land. (7)

2.3 Thingy serien

Nordic har designet en rekke utviklingskort og prototypplattformer, deriblant Nordic Thingy:91 og Thingy:52 som gjenspeiler mikroprosessorene nRF9160 og nRF52832. Begge disse prototypplattformene har en rekke sensorer for bevegelse og luftkvalitet. Thingy:91 er utstyrt med LTE modem for NB-IoT og LTE-M, samt Bluetooth Low Energy (BLE) og GPS mottaker. Thingy:52 støtter derimot ikke LTE eller GPS, men er i stedet utstyrt med noen ekstra sensorer. Thingy kortene er best egnet til Proof-of-Concept da disse ikke har innebygget debugger. De har derimot et kompakt design med Li-Po batteri. (8) (9)

Figur 2 – Thingy:91

Utviklingskortene nRF9160dk og nRF52840dk har J-Link debugger som gjør det kjapt og enkelt å laste kode, samt feilsøking under utvikling av programvare. Programvarekoden man designer for et kort kan enkelt flyttes over til å gå på annen hardware, gitt at hardwaren har tilsvarende funksjonalitet. Utviklingskortene har ikke innebygde sensorer, de tilbyr kun simulerte verdier for sensorene.

2.4 nRF9160

nRF9160 har da altså innebygget LTE modem kombinert med GPS og en applikasjonsprosessor. Prosessoren er en 32 bit ARM Cortex – M33 på 64MHz med 1MB flash og 256kB RAM. Av I/O kan man nevne 12bit ADC, RTC (Real Time Clock), SPI, I^2C, UART, PWM modulator med mer. Chipen har 32 GPIO-pinner. Modemet støtter NB-IoT og LTE-M med TCP/UDP og TLS sikkerhet. nRF9160 har ikke Bluetooth. (10) (11) (12)

Figur 3 – nRF9160 SiP

2.5 nRF52 serien

nRF52840 er en SoC beregnet til lokale IoT oppgaver over BLE, mikrokontrolleren er basert på en ARM Cortex – M4F prosessor med tilsvarende egenskaper som den brukt i nRF9160. Det finnes mange utgaver av denne serien med SoCer, nRF52832 er en enklere utgave med mindre minne og færre GPIO pinner. Disse mikrobrikkene støtter et bredt utvalg av trådløse standarder i 2.4GHz spekteret, deriblant Bluetooth MESH, Thread, Zigbee, ANT med mer. Ved å kombinere denne med nRF9160 vil man kunne utvikle komplekse systemer med både LTE, GPS og BLE. (13) (14)

Figur 4 – nRF52840 SoC

2.6 Programvareverktøy

Nordic har lagt opp til at man benytter SEGGER Embedded Studio til utvikling av programkode for mikrobrikkene deres. SEGGER er gratis for personlig bruk, samt for utvikling av kode til enkelte mikrobrikker. Alle brikkene til Nordic er inkludert i denne ordningen. SEGGER kan installeres på plattformene Windows, Linux og Mac. (15)

Nordic tilbyr en rekke hjelpemidler for utvikling av IoT enheter, disse tingene er samlet i programmet nRF Connect. Her finner man blant annet LTE Link Monitor, Bluetooth Low Energy verktøy, Programmerings-verktøy, Power Profiler og Toolchain Manager. Toolchain Manager er SEGGER, programmet lastes ned og installeres direkte fra nRF Connect. Inkludert i denne nedlastingen er også demoprosjekter og driverkode for de aktuelle utviklingskortene for denne rapporten. (16)

I neste kapittel vil vi ha behov for nRF Connect og SEGGER Embedded Studio (Toolchain Manager), start gjerne nedlasting og installasjon av dette allerede nå.
Versjonen jeg har brukt er nRF Connect SDK v1.5.1.
nRF Connect: https://www.nordicsemi.com/Products/Development-tools/nrf-connect-for-desktop
Merk: "Knapp" for nedlasting er teksten "*nrfconnect-setup-3.7.0-ia32.exe*".

2.7 Zephyr RTOS

I IoT sammenheng begynner systemene å bli såpass komplekse at det er nødvendig med en form for RTOS (Real Time Operating System). Tanken bak et slikt system er å fordele prosessortid på flere prosesser som nærmest kan gå i parallell. I Zephyr kalles slike prosesser for *Threads*. Når man jobber med sanntidssystemer, typisk ting som er koblet i nettverk, er det viktig at prosessoren ikke blir stående med en oppgave i lengre tid. Tenk på det som multi-tasking. Dersom du vil lære mer om sanntidsprogrammering kan det leses her: (17)

Nordic har lagt opp til å benytte Zephyr OS som operativsystem. Zephyr Project er del av The Linux Foundation og utvikles i samarbeid med flere store aktører. Det er lagt opp til at systemet skal være skalerbart etter hva man har behov for. Zephyr OS, samt FreeRTOS, er blant de meste brukte åpen-kildekodesystemene for bruk i IoT sammenheng. (18) (19) (20)

2.8 Dokumentasjon

Nordic har gjort en omfattende jobb med å dokumentere produktene sine, dokumentasjonen er delt inn i tre hoveddeler.

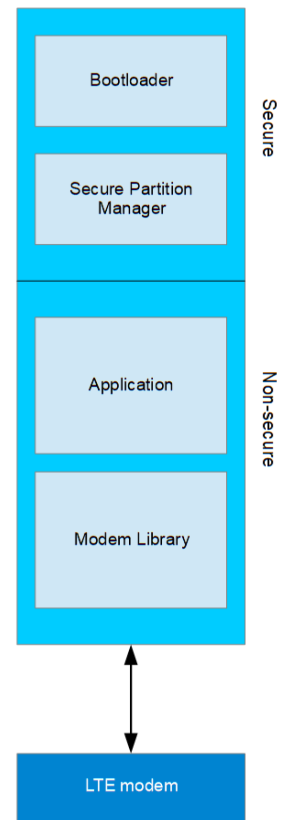
- Infosenter – oversikt over dokumentasjon, hardware og diverse revisjoner.
  URL: https://infocenter.nordicsemi.com
- nRF Connect SDK – oversikt over utviklingsverktøy, programvarebiblioteker og drivere.
  Merk: Nederst på venstre side kan man bytte mellom ulike deler av denne dokumentasjonen.
  URL: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/index.html
- DevZone – forum med guider og muligheter for diskusjon.
  Merk: Nybegynnerguider finnes i menyen under "*Nordic content*" -> "*nRF Connect SDK guides*".
  URL: https://devzone.nordicsemi.com/
- Dokumentasjon for Zephyr finnes her:
  URL: https://docs.zephyrproject.org

3. Kode - Grunnleggende

Vi vil nå begynne å se på hvordan vi kan utvikle egen programvare til mikrobrikkene. For å gjøre dette på en effektiv måte er det viktig å forstå oppbygningen av koden som Nordic allerede har utviklet. For å forstå de grunnleggende tingene starter vi med eksempelprosjektet "*Threads*". Vi skal straks se på hva threads er og hvordan det brukes.

Det er nå forventet at nRF Connect og SEGGER (Toolchain Manager) er installert. For å hente frem eksempelprosjekter finner man disse under "*Files*" -> "*Open nRF Connect SDK Project…*". Her finner man blant annet prosjektet "*Threads*" og kortet "*nrf9160dk_nrf9160ns*".

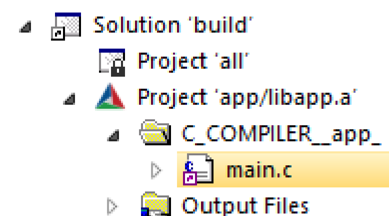
Prosessorkjernen for applikasjoner (MCU) er i dette tilfellet Cortex-M33 i nRF9160. Den er delt inn i to deler, en sikker og en usikker del. Programvaren vi designer skal kjøres i den usikre delen, derav navnet *nrf9160ns*. (21) Se figur 5. For å lese mer om dette, besøk siden: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/ug_nrf9160.html#available-drivers-libraries-and-samples.



Figur 5 – nRF9160 arkitektur

Prosjektet "*minimal*" er et tomt prosjekt som er egnet for å teste ut kodesnutter underveis. Prosjektet inneholder en del grunnleggende konfigurering og selvfølgelig Zephyr RTOS. Når man åpner et slikt prosjekt redigerer man direkte i de kodefilene som ble lastet ned i forbindelse med installasjon. Det kan by på utfordringer å finne igjen de opprinnelige filene, det er dermed lurt å ta en kopi av de prosjektene du har planer om å redigere. Ved å høyreklikke på "*main.c*" kan man åpne mappa i Windows filutforsker. Se figur 6.

Det kan også være praktisk å åpne kodefiler i Notepad++ dersom du har behov for å se på kode utenfor det prosjektet du jobber i. SEGGER kan bare operere med et prosjekt av gangen.



Figur 6 – Filstruktur

3.1 Threads

En tråd (*thread*) er et objekt i Zephyr kjernen som blir brukt for å utføre prosesser som er for lange og komplekse til å bli utført av et interrupt. Hver thread får tildelt sitt område i arbeidsminne (RAM), mengden RAM avgjør hvor mange threads et system kan håndtere. I sanntidssystemer går prosesser i parallell, de prosessene vi skal designe skrives i form av funksjoner. Når man ønsker kjørt en slik funksjon velger man hvordan den skal kalles. Den kan enten kalles via en thread, fra workqueue eller den kan kalles fra main-løkka. Workqueue brukes for prosesser som ikke haster, det er en egen thread som tar unna elementene som ligger i workqueue. Main-løkka har den høyeste prioriteten i systemet, prioritet 0. (22)

Les mer her:

- Threads: docs.zephyrproject.org/latest/reference/kernel/threads/index.html
- Workqueue: docs.zephyrproject.org/latest/reference/kernel/threads/workqueue.html
- Scheduling: docs.zephyrproject.org/latest/reference/kernel/scheduling/index.html

3.1.1 Opprette en *thread*

Når en thread skal deklareres velger man en passende størrelse på arbeidsminne (STACKSIZE) og hvilken prioritet den skal kjøres i. Selve deklarasjonen er en enkelt kodelinje, **K_THREAD_DEFINE**(…). Figur 7 viser all kode som trengs for å deklarer og kjøre en thread i Zephyr RTOS. Rutinen **k_msleep**(…) legger den aktuelle threaden i dvale i en gitt tid. Hele systemet kan legges i dvale dersom det ikke er andre threads som kjøres.

```
/* -----  Thread   ----- */
#include <zephyr.h>

/* ----- Constants ----- */
#define STACKSIZE 1024
#define PRIORITY 7

/* ----- Function  ----- */
void printk_funk(void)
{
        while(1)
        {
                printk("Thread is running \n");
                k_msleep(5000);
        }
}

/* ----- Define Thread ----- */
K_THREAD_DEFINE(funk_id, STACKSIZE, printk_funk,
                NULL, NULL, NULL, PRIORITY, 0, 0);
```

Figur 7 – Deklarere en thread

Det kan ofte være lurt å lese mer om en forhåndsdefinert makro, dette gjøres ved å høyreklikke på den og velge "*Go To Definition*". Vi ser i figur 8 at makroen skal ha 9 inngangsparametere. Kommentarene gir en god pekepinn på hvordan makroen er ment å brukes.

```
 * @param name Name of the thread.
 * @param stack_size Stack size in bytes.
 * @param entry Thread entry function.
 * @param p1 1st entry point parameter.
 * @param p2 2nd entry point parameter.
 * @param p3 3rd entry point parameter.
 * @param prio Thread priority.
 * @param options Thread options.
 * @param delay Scheduling delay (in milliseconds), zero for no delay.
 */
#define K_THREAD_DEFINE(name, stack_size,                         \
                        entry, p1, p2, p3,                        \
                        prio, options, delay)                     \
    K_THREAD_STACK_DEFINE(_k_thread_stack_##name, stack_size);    \
    struct k_thread _k_thread_obj_##name;                         \
    Z_STRUCT_SECTION_ITERABLE(_static_thread_data, _k_thread_data_##name) =\
          Z_THREAD_INITIALIZER(&_k_thread_obj_##name,      \
                _k_thread_stack_##name, stack_size,        \
                entry, p1, p2, p3, prio, options, delay,   \
                NULL, name);                               \
        const k_tid_t name = (k_tid_t)&_k_thread_obj_##name
```

Figur 8 – "Go to definition"

3.1.2 Workqueue

En workqueue er en FIFO (first-in-first-out) liste over jobber man ønsker utført. Workqueue benytter en egen dedikert thread for å få utført disse jobbene. Threaden har vanligvis lavere prioritet enn andre threads i systemet. Vanligvis brukes workqueue i sammenheng med ISR (Interrupt Service Routine) eller en høy prioritets thread der man ikke har tid til å gjøre prosesseringen umiddelbart. Som med threads definerer man størrelsen på arbeidsminne og hvilken prioritet man ønsker. (23) (24)

Videre blir disse to strukturene definert:

struct k_work er for arbeidselementet som skal legges i køen (25)
struct k_work_q definerer selve køen som workqueue bruker (26)

Deretter starter man workqueue prosessene:

k_work_init initialiserer workqueuen for funksjonen *print_workqueue*.
k_work_q_start starter workqueuen
k_work_submit legger arbeidselementet til i workqueue køen.
k_msleep(2500) setter systemet i dvale i 2500ms.

```
/* ----- Workqueue ----- */
#include <zephyr.h>

/* ----- Constants ----- */
#define STACKSIZE 512
#define PRIORITY 5

K_THREAD_STACK_DEFINE(stackarea, STACKSIZE);

/* ----- Function  ----- */
void print_workqueue(struct k_work *work)
{
    printk("Workqueue is running \n");
}

void main(void)
{
/* ----- Workqueue ----- */
    struct k_work work;
    struct k_work_q my_work_q;
    k_work_init(&work, print_workqueue);
    k_work_q_start(&my_work_q, stackarea, /*New-line*/
          K_THREAD_STACK_SIZEOF(stackarea), PRIORITY);

    while(1)
    {
        k_work_submit(&work); //Add item to workqueue
        k_msleep(2500);
    }
}
```

Figur 9 – Workqueue

3.1.3 Styre LEDs

Videre ser vi på hvordan en lysdiode konfigureres, det vil her være LED1 på kortet som blinker med en frekvens på 1Hz. Det må inkluderes driver for GPIO samt Devicetree for dette prosjektet. Devicetree (DT) er en datastruktur over hardware som operativsystemet bruker for å tilpasse seg den hardwaren det kjører på. Dette gjør at koden du programmerer kan kjøres på forskjellige hardware-konfigurasjoner. Det er gjerne produsenten av hardwaren som generer DT-filene. (27) (28) (29)

```
 1 /* ----- Blinky ----- */
 2 #include <zephyr.h>
 3 #include <devicetree.h>
 4 #include <drivers/gpio.h>
 5
 6 //Devicetree node identifier for the "led0" alias
 7 //led0 in DT is LED1 on the board
 8 #define LED1_NODE DT_ALIAS(led0)
 9
10 //Check and configure of led1.
11 #if DT_NODE_HAS_STATUS(LED1_NODE, okay)
12 #define LED1    DT_GPIO_LABEL(LED1_NODE, gpios)
13 #define PIN     DT_GPIO_PIN(LED1_NODE, gpios)
14 #define FLAGS   DT_GPIO_FLAGS(LED1_NODE, gpios)
15
16 #else // led1 is not defined for this board
17 #error "Unsupported board"
18 #endif
19
20 void main(void)
21 {
22     const struct device *dev;
23     bool led_is_on = true;
24     int ret;
25
26     dev = device_get_binding(LED1);
27     if (dev == NULL) //Error check
28         return;
29
30     ret = gpio_pin_configure(dev, PIN, GPIO_OUTPUT_ACTIVE | FLAGS);
31     if (ret < 0) //Error check
32         return;
33
34     while (1)
35     {
36         gpio_pin_set(dev, PIN, (int)led_is_on);
37         led_is_on = !led_is_on;
38
39         k_msleep(500);
40     }
41 }
```

Figur 10 – Blinky

Figur 10 viser nødvendig kode for å blinke med en lysdiode. Linje 8 viser hvordan man bruker devicetree for å hente node-id til led0. Videre er linje 11 til 18 for å verifisere at det er funnet gyldig node-id og for å konfigurere lysdioden med rett pinout i forhold til hardware koden kjører på. Så brukes **device_get_binding**() for å gi en peker til objektet vi ønsker å kontrollere, LED1. Linje 30 konfigurerer pinnen som utgang og setter den høy (aktiv). **gpio_pin_set**() setter verdi på utgangen, denne vil skru lysdioden av og på annenhver gang i dette eksempelet.

3.1.4 Oppsummering

Vi samler nå de tre kodesnuttende i et felles prosjekt og ser hvordan det virker. For å ikke rote til demoprosjektene bruker vi prosjektet "*Minimal*". Bygg nå prosjektet og se at kompileringen går gjennom uten feil. For å laste kode på kortet går du til "*Target*" -> "*Download zephyr/zephyr.elf*".

For å se utskriftene i terminalvinduet går du til "*Tools*" -> "*Terminal Emulator*". Sett "*Baud Rate*" til 11520 og prøv deg frem til hvilken COM-port som gjelder for ditt system. nRF9160dk har 3 UART grensesnitt gjennom virtuelle COM-porter. To porter til nRF9160 og en til nRF52840. Terminalvinduet dukker opp med et telefonikon i vinduet nederst på skjermen. (30)

I figur 12 ser vi resultatet av sammenslåingen. Dette er på ingen måte ideell kode da det blant annet er lagt en uendelig løkke i workqueue. Dette vil oppta køen og hindre eventuelle andre elementer fra å kjøre. Allikevel gir det en god indikasjon på hvordan et enkelt prosjekt kan se ut. Vi har 3 prosesser som går i parallell. Merk at denne koden har "while(1)" på tre forskjellige steder, og alle prosessene går uansett.

I figur 11 ser vi utskrift fra terminalvindu. Dette er en god illustrasjon på hvordan kode kjøres i et RTOS. Prosessene går i parallell og dette fører til at de kan gå i beina på hverandre. Vi ser at utskriften fra Threads blir avbrutt for å kjøre Workqueue, og motsatt. I komplekse applikasjoner kan denne typen feil være omtrent umulig å finne ut av i ettertid. Det er viktig å tenke nøye gjennom hvilke prosesser som kan skape utfordringer dersom de blir avbrutt. En vanlig tabbe er prosesser som leser/skriver til samme minneområde. For å håndtere slike problemstillinger kan man benytte semaforer, spinlock eller deaktivere interrupt. Dette vil vi komme tilbake til i et senere kapittel, men du kan også lese om det her: https://www.geeksforgeeks.org/g-fact-70/

```
Workqueue is running
Thread is runniWorkqueue is running
ng
Workqueue is running
Thread is running Workqueue is running

Workqueue is running
Thread is running
Workqueue is running
Workqueue is running
Thread is running
```

Figur 11 – Fra terminal

```c
/* ----- Understand the basics ----- */
#include <zephyr.h>
#include <devicetree.h>
#include <drivers/gpio.h>

/* ----- Constants ----- */
#define STACKSIZE_thread 1024
#define PRIORITY_thread 7
#define STACKSIZE_work_q 512
#define PRIORITY_work_q 5

K_THREAD_STACK_DEFINE(stackarea, STACKSIZE_work_q);

/* ----- Blinky ----- */
//Devicetree node identifier for the "led0" alias
//led0 in DT is LED1 on the board
#define LED1_NODE DT_ALIAS(led0)

//Check and configure of led1.
#if DT_NODE_HAS_STATUS(LED1_NODE, okay)
#define LED1    DT_GPIO_LABEL(LED1_NODE, gpios)
#define PIN     DT_GPIO_PIN(LED1_NODE, gpios)
#define FLAGS   DT_GPIO_FLAGS(LED1_NODE, gpios)

#else // led1 is not defined for this board
#error "Unsupported board"
#endif

/* ----- Functions  ----- */
void print_thread(void)
{
    while(1)
    {
        printk("Thread is running \n");
        k_msleep(5000);
    }
}

void print_workqueue(struct k_work *work)
{
    while(1)
    {
        printk("Workqueue is running \n");
        k_msleep(2500);
    }
}

/* ----- Define Thread ----- */
K_THREAD_DEFINE(funk_id, STACKSIZE_thread, print_thread, NULL, NULL, NULL, PRIORITY_thread, 0, 0);

/* ------------------------------ */
/* ---------- Main-loop ---------- */
/* ------------------------------ */
void main(void)
{
  /* ----- Workqueue ----- */
    struct k_work work;
    struct k_work_q my_work_q;
    k_work_init(&work, print_workqueue);
    k_work_q_start(&my_work_q, stackarea, K_THREAD_STACK_SIZEOF(stackarea), PRIORITY_work_q);

    k_work_submit(&work); //Add item to workqueue

  /* ----- Blinky ----- */
    const struct device *dev;
    bool led_is_on = true;
    int ret;

    dev = device_get_binding(LED1);
    if (dev == NULL) //Error check
      return;

    ret = gpio_pin_configure(dev, PIN, GPIO_OUTPUT_ACTIVE | FLAGS);
    if (ret < 0) //Error check
      return;

    while(1)
    {
        gpio_pin_set(dev, PIN, (int)led_is_on);
        led_is_on = !led_is_on;
        k_msleep(500);
    }
}
```

Figur 12 – Understanding the basics

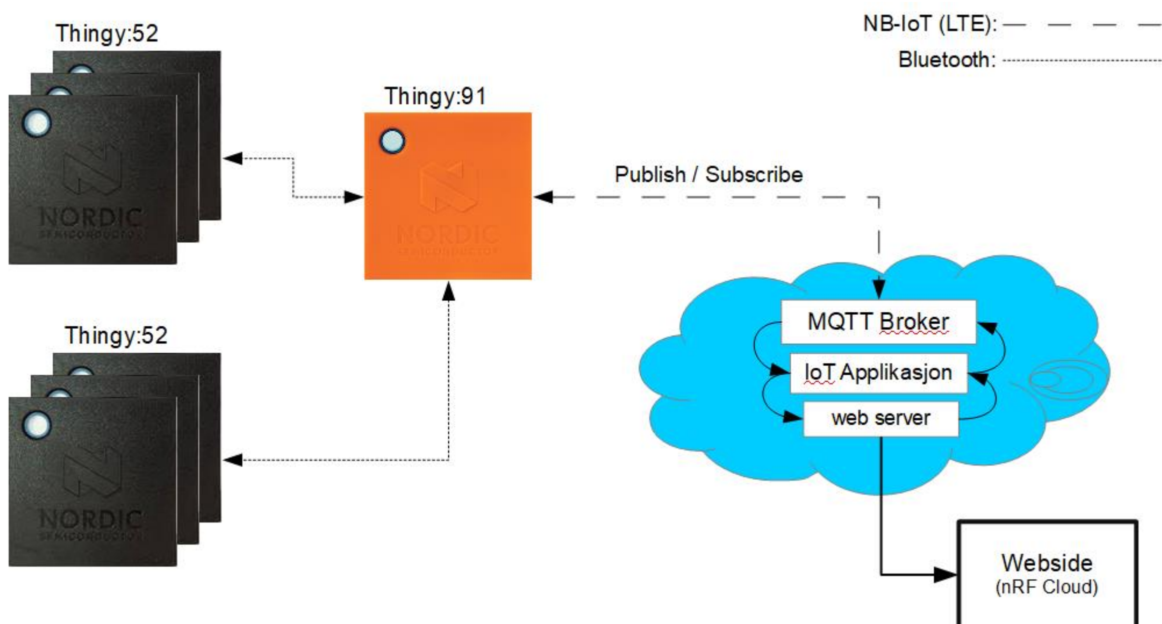
4. Kode – Opprette forbindelse til sky

Vi vil nå begynne å se på hvordan man kommuniserer med en skytjeneste. Vi starter med å snakke litt om temaet og hvordan Nordic har sett for seg at det skal gjøres i deres eksempler. Videre ser vi på koden i eksempelet "*cloud_client*" for "*nrf9160dk_nrf9160ns*". Vi vil så utvide koden slik at vi kan styre utviklingskortet fra en nettleser via internett og NB-IoT standarden.

NB-IoT (*Narrowband Internet of Things*) er en av protokollene innenfor LTE standarden. Det er utviklet for systemer som er strømgjerrige og ikke har krav til høy båndbredde. Maks hastighet over NB-IoT er 127kbit/s for nedlasting og 159kbit/s i opplasting over halv duplex. Det kan også være en betydelig forsinkelse på opptil 10 sekunder. For applikasjoner som krever høyere båndbredde kan man benytte LTE-M. Fordelene her er mye lavere forsinkelse og mulighet for taleoverføring. (31) (32) (33)

NB-IoT gjør det mulig å utvikle kostnadseffektive enheter som kan holdes aktive over lange tidsintervall ved hjelp av batteri og/eller energihøsting. Protokollen støtter PSM (Power Saving Mode) som blant annet tillater å holde en forbindelse aktiv selv om den tilkoblede enheten ligger i dvale. Enheten kan våkne opp, sende en melding for så å legge seg i dvale igjen, dette uten å måtte opprette en ny forbindelse med basestasjonen. Det vil ikke være mulig å sende data til enheten når den er i dvale, men dersom basestasjonen støtter det kan den i midlertid mellomlagre meldingen så enheten kan hente dem når den er våken. Ved relevant etterarbeid kan jeg anbefale å lese mer om NB-IoT standarden her (34).

I mange tilfeller er det gunstig å bruke MQTT (Message Queueing Telemetry Transport) for dataoverføring i NB-IoT nettverk, dette er særlig fordi MQTT krever svært liten header til dataoverføringen. MQTT kan klare seg med kun 2 byte i motsetning til http som trenger 26 byte. Som med alle protokollene i applikasjonslaget kreves det en underliggende kommunikasjonskanal, MQTT krever at det benyttes TCP/IP ut på internett. Dersom det er behov for oppfriskning av sammenhengen mellom de ulike lagene i et datanettverk kan det leses mer om OSI modellen i kapittelet "*Kjekt å vite*" mot slutten av rapporten.



Figur 13 – IoT med Nordic Semiconductor

For dette prosjektet vil vi benytte skyløsning fra Nordic Semiconductor. Nordic designet nRF Cloud for å illustrere funksjonaliteten til prosjektet "*asset_tracker*" (35) på en god måte. Dette gikk dessverre ut over tilpasningsmulighetene, og gjør tjenesten lite anvendelig for bruk i andre sammenhenger. Vi kommer hovedsakelig til å benytte terminalvinduet for å illustrere kommunikasjonen i vårt prosjekt. Skyløsningen er basert på Amazon Web Services (AWS) som er mye brukt i IoT sammenheng. Du kan lese mer om AWS IoT her (36) (37).

For å komme i gang med nRF Cloud må du opprette en bruker. Når du så har fått logget deg inn må utviklingskort og SIM-kort registreres på brukeren. Guide for dette finnes her: nrfcloud.com og eventuelt videoguide her https://youtu.be/rMvW3HXLErk?t=575 (YouTube kanalen til Nordic).

Figur 13 viser hvordan man kan se for seg et IoT sensornettverk basert på prototypenheter fra Nordic. Her vil man benytte Thingy:52 som noder i nettverket som kommuniserer med en sentral, Thingy:91, som igjen kommuniserer med en web server over NB-IoT over internett. Jeg kommer ikke til å ta for meg nodene i denne rapporten, fokus her ligger på kommunikasjon fra nRF9160dk (Thingy:91) til nRF Cloud.

4.1 MQTT og JSON

Kommunikasjonen som går mellom sentralen og nRF Cloud er MQTT meldinger. Det er krav om at nyttelasten i disse meldingene er på JSON-format. JSON (JavaScript Object Notation) er mye brukt i web-applikasjoner for å overføre data i tekst-format. En av fordelene med JSON fremfor XML er at det er plassbesparende. Der XML har dobbelt sett med start- og slutt-notasjoner har JSON bare start-notasjon. Med andre ord sparer man båndbredde ved å benytte JSON.

Støttede datatyper er blant annet tekst, tall, boolske verdier og tabeller. Data separeres med komma, krøllparenteser brukes for å avgrense objekter og klammeparenteser brukes for array. Et eksempel på datastruktur er vist i figur 14.

Figur 14 – JSON

MQTT protokollen ble utviklet i 1999 for å være en enkel og strømbesparende protokoll som kunne brukes over nettverk med lav båndbredde. Den ble først tatt i bruk for å overvåke rørledninger i olje- og gassindustrien. Protokollen har i senere tid blitt sett svært praktisk i M2M og IoT sammenhenger på grunn av sine fordeler med støtte for mange sensorer over en nettlinje med lav båndbredde. I utgangspunktet er MQTT protokollen er avhengig av TCP/IP protokollen, men en utvidelse kalt MQTT-SN (MQTT for Sensor Networks) gjør det mulig å benytte Bluetooth som transmisjonsmedium. (38) (39) (40)

I et MQTT nettverk har man noen sentrale begreper som det er viktig å forstå, se figur 15 i sammenheng med følgende liste.

- Klient: Publish/Subscribe – Sender eller mottar data
- Server: MQTT Broker – Mottar data fra klienter og sender videre ut til alle som lytter
- Emne/type data: Topics – Hvor data skal publiseres eller hvor du skal lytte
- Meldinger: Messages – Nytteverdien i dataene som går på linja (Payload)

MQTT Broker opptrer som en server. Ute i nettverket har man et antall klienter som sender data til serveren. Klientene kan abonnere på forskjellig typer informasjon, på denne måten kan serveren sende ut en melding til mange klienter samtidig. MQTT Broker har ansvaret for å motta alle meldinger, behandle dem og publisere i riktig topics.

Meldingene skal ha en definert QoS (Quality of service), det vil si hvordan man skal forsikre seg om at meldingen blir mottatt. Valgene man har er: Maks en gang, minst en gang eller nøyaktig en gang. Serveren kan også mellomlagre den siste meldingen den mottok fra en klient for å ha mulighet til å sende denne til andre enheter som kobler seg på i ettertid. Man kan også definere hva som skal skje dersom en klient dør. For å forstå QoS bedre vil jeg anbefale å ta en kjapp titt her: (41)
Last Will og *Testament* er forklart her: (42)

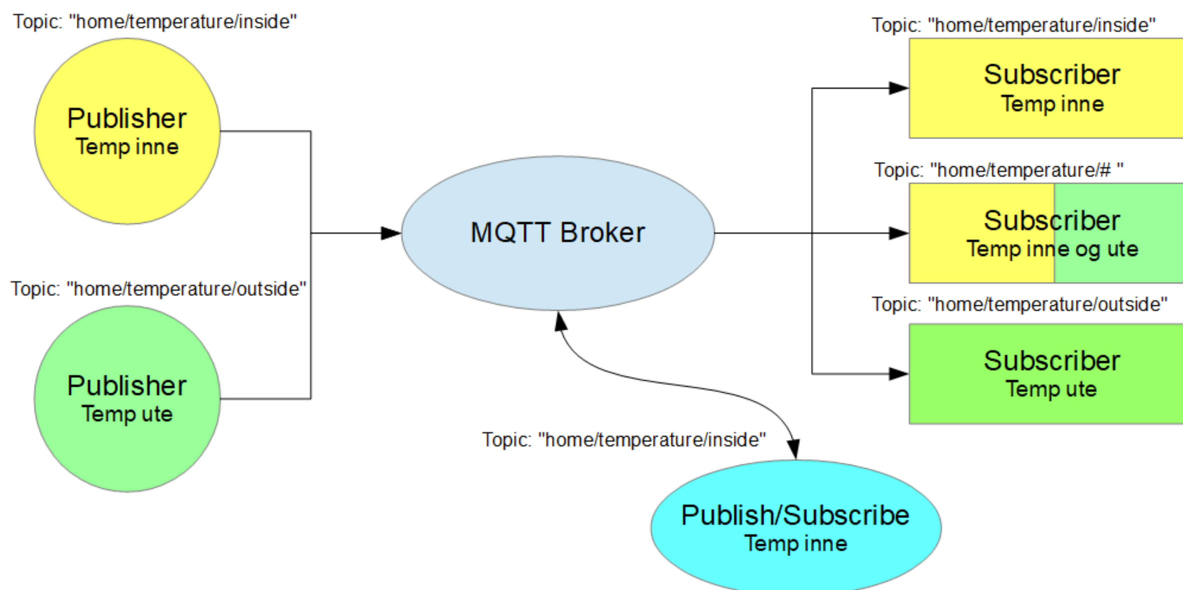
Topics forteller hvordan du ønsker at data skal registreres i systemet, hvordan data sorteres er avhengig av hvordan det er praktisk å sortere dem. I et hus har man typisk flere sensorer i hvert rom og en felles sentral, fra sentralen ønsker man å skaffe oversikt over disse sensordataene. Fra sentralen kan man også ha ønske om å skru alt lys av eller på. Sensorene kan da typisk sorteres etter rom i huset. Merk at *topics* er case-sensitiv, det er forskjell på å skrive OFFICE og office!

home/kitchen/light | home/kitchen/temperature
home/office/ light | home/office/ temperature

Ønsker man å få alle temperaturer i systemet kan man benytte " # " som universaltegn
home/#/temperature

Du kan lese mer om MQTT her:

- https://thenewstack.io/mqtt-protocol-iot/
- https://randomnerdtutorials.com/what-is-mqtt-and-how-it-works/



Figur 15 – MQTT forklart

4.2 Cloud_client

Åpne nå "*cloud_client*" for "*nrf9160dk_nrf9160ns*" fra "*File*" -> "*Open nRF Connect SDK Project…*". Det er en feil i dette prosjektet som gjør at det ikke er mulig å kompilere. Feilen er diskutert i DevZone og med henvisning til en såkalt "*pull-request*" på GitHub. I korte trekk er det ikke satt av nok plass til Secure Partision Manager (SPM). Dette fører til feilmeldingen SPM Flash overflow ved kompilering. (43) Filene som skal endres finner dere her: https://github.com/nrfconnect/sdk-nrf/pull/4103/files, og de ligger lokalt på din maskin her:

- (USERNAME) \ncs\v1.5.1\nrf\subsys\spm – Kconfig
- (USERNAME) \ncs\v1.5.1\nrf\samples\spm – prj.conf
- (USERNAME) \ncs\v1.5.1\nrf\samples\spm – prj_minimal.conf (denne skal opprettes)

Etter å ha ordnet opp i dette må det kjøres en "*CMake*" for at kompileringen skal ta med de nye konfigurasjonsfilene. Dette gjøres enkelt fra "*Project*" -> "*Run CMake*" dersom man blir informert om at dette er nødvendig. Det tok meg en ukes tid å finne ut av dette… Takk til Irene Sollie fra Nordic som til slutt guidet meg gjennom denne prosessen!

Da bør alt være klart for å laste kode på kortet og se at man får kontakt med nRF Cloud. Det kan ta et par minutter for kortet å koble seg opp, man kan følge oppkoblingen i terminal i SEGGER. Det sendes melding til nRF Cloud hver gang man trykker på "*Button 1*" på kortet, figur 16.


```
"Received" : { 1 item
  "state" : { 1 item
    "reported" : { 1 item
      "message" : "Hello Internet of Things!"
    }
  }
}
```


Figur 16 – Cloud message

4.3 Kode for oppkobling

Vi starter kodeanalysen ved å se på funksjonen *main* (figur 17), dette gir oss et godt bilde av hvordan gangen i koden er. Du oppdager fort at det er mange nye makroer i koden, vi skal gå gjennom alle sammen men jeg anbefaler uansett å besøke dem via "*Go to definition*". Det er også en del konfigurasjon i et slikt system, mange av funksjonene settes opp via et eget konfigurasjonsverktøy. Dette finner man under "*Project*" -> "*Configure nRF Connect SDK Project…*".

Konfigureringsverktøyet er todelt, vi kommer bare til å ta for oss "*menuconfig*". Verktøyet brukes for å aktivere eller deaktivere funksjonalitet i systemet, det kan være praktisk å søke etter de aktuelle funksjonen ved å benytte "*Filter*" boksen øverst i vinduet. Jeg har erfart at konfigureringsverktøyet kan være ustabilt og henge seg opp under visse forhold, dette kommer jeg tilbake til litt senere.

Som standard er det LTE-M som benyttes som kommunikasjonsprotokoll for prosjektet "*cloud_client*". Hvilken protokoll som benyttes endrer man i konfigureringsverktøyet. Søk etter "*LTE*" så vil lista over protokoller dukke opp under "*Select network mode*". Igjen takk til Irene Sollie som hjalp meg å få konfigurert dette.

Jeg dro hjem til Nøtterøy mot slutten av bachelorprosjekt-perioden. Etter at jeg kom hjem har ikke NB-IoT virket, jeg har derfor benyttet LTE-M for siste del av prosjektet. Dette skyldes trolig dårlig dekning i området jeg bor, men uten at jeg kan si det med 100% sikkerhet. Det utgjør ingen praktisk betydning for programmeringens del, men følte allikevel det er greit å informere om dersom andre har problemer med at NB-IoT ikke fungerer optimalt.

```
247 void main(void)
248 {
249         int err;
250         LOG_INF("Cloud client has started");
251
252         cloud_backend = cloud_get_binding(CONFIG_CLOUD_BACKEND);
253         __ASSERT(cloud_backend != NULL, "%s backend not found", CONFIG_CLOUD_BACKEND);
254
255         err = cloud_init(cloud_backend, cloud_event_handler);
256         if (err)
257             LOG_ERR("Cloud backend could not be initialized, error: %d", err);
258
259         work_init();
260         modem_configure();
261
262 #if defined(CONFIG_CLOUD_PUBLICATION_BUTTON_PRESS)
263         err = dk_buttons_init(button_handler);
264         if (err)
265             LOG_ERR("dk_buttons_init, error: %d", err);
266
267 #endif
268         LOG_INF("Connecting to LTE network, this may take several minutes...");
269
270         k_sem_take(&lte_connected, K_FOREVER);
271
272         LOG_INF("Connected to LTE network");
273         LOG_INF("Connecting to cloud");
274
275         k_delayed_work_submit(&connect_work, K_NO_WAIT);
276 }
```

Figur 17 – Cloud_Client - main

LOG_INF er første makro som dukker opp i *main* og benyttes for å loggføre systemhendelser. Man kan kategorisere hendelsene etter hva de er, om de er til informasjon (**LOG_INF**), advarsler (**LOG_WRN**) eller feil (**LOG_ERR**). For å benytte funksjonen må "*logging/log.h*" inkluderes og **LOG_MODULE_REGISTER**(name, logging_level); må kjøres for initialisering. I tillegg må "Logger" aktiveres i konfigurasjonsfilene. Mer om loggføring kan leses her, merk at *Logger API* inneholder fullstendig liste over kategorier. (44)

__ASSERT brukes for å melde fra om feil, den sjekker om argumentet er usant, hvis det er tilfelle vil den skrive ut definert feilmelding og sette systemet i en uendelig loop. Som med **LOG_INF** må denne også aktiveres i konfig, søk etter assert. (45)

Merk at makroer skilles fra funksjoner ved at de skrives med store bokstaver.

cloud_get_binding er en funksjon som gir en peker til den registrerte skyløsningen i systemet. **cloud_init** initialiserer skyløsningen og gjør all nødvendig konfigurasjon. Første parameter er peker til skyløsningen og andre parameter er en peker til en "callback" funksjon. Det vil si en funksjon som blir trigget når det skjer en hendelse. Figur 18 viser eksempler på hendelser. Vi vil gå nærmere inn på dette når vi tar for oss sending og mottak av data om et lite øyeblikk.

work_init legger til to elementer i workqueue, disse har med skykoblingen å gjøre.

modem_configure konfigurerer modemet.

#if **defined** er i praksis #ifdef, men med støtte for flere betingelser. Hvis betingelsene er forskjellig fra NULL vil denne være sann.

```
 86 void cloud_event_handler(const struct cloud_backend *const backend,
 87                          const struct cloud_event *const evt,
 88                          void *user_data)
 89 {
 90         ARG_UNUSED(user_data);
 91         ARG_UNUSED(backend);
 92 
 93         switch (evt->type) {
 94         case CLOUD_EVT_CONNECTING:
 95                 LOG_INF("CLOUD_EVT_CONNECTING");
 96                 break;
 97         case CLOUD_EVT_CONNECTED:
 98                 LOG_INF("CLOUD_EVT_CONNECTED");
 99                 cloud_connected = true;
100                 break;
101         case CLOUD_EVT_READY:
102                 LOG_INF("CLOUD_EVT_READY");
103                 k_delayed_work_cancel(&connect_work);
104         case CLOUD_EVT_DATA_SENT:
105                 LOG_INF("CLOUD_EVT_DATA_SENT");
106                 break;
107         case CLOUD_EVT_DATA_RECEIVED:
108                 LOG_INF("CLOUD_EVT_DATA_RECEIVED");
109                 LOG_INF("Data received from cloud: %.*s",
110                         evt->data.msg.len,
111                         log_strdup(evt->data.msg.buf));
112                 break;
```

Figur 18 – Cloud_Client - cloud_event_handler

Jeg vil nå beskrive gangen i oppkoblingsprosessen, dette innebærer å hoppe mellom kodelinjer i hele *main.c* og jeg vil derfor henvise til nummer på linjene fremfor å ta med kodesnutter direkte. *main.c* finnes som vedlegg til denne rapporten, fila har navnet *cloud_client.c* i Programkode.zip mappen.

248 – Variabelen *err* brukes for returverdi fra funksjoner.
252 – *cloud_backend* er peker til skyløsningen.
253 – Hvis pekeren er NULL får vi feilmelding og systemet settes i uendelig loop.
256 – Funksjonen *cloud_init* blir kjørt, dersom *err* ikke er NULL har det har oppstått en feil.
Denne vil i sin tur starte *cloud_event_handler* som ser etter hendelse fra skyløsning.
262 – Funksjonen *work_init* blir kjørt.
152 – *cloud_update_work_fn* blir initialisert til en forsinket workqueue.
153 – *connect_work_fn* blir initialisert til en forsinket workqueue. (46)
263 – Konfigurerer modemet.
208 – Her kunne man satt opp PSM (Power saving mode) ved behov.
223 – Dette blir funksjonskall etter funksjonskall for oppsett av LTE forbindelsen.
lte_handler blir etterhvert kalt.
168 Semaforen *lte_connected* blir frigitt.
266 – Konfigurer *button1* i ekstern fil.
240 – Ved knappetrykk blir adressen til funksjonen *cloud_update_work* lagt i workqueue.
273 – Venter til semaforen er frigitt fra kodelinje 168. Når den blir det vil LTE oppkoblingen være vellykket og programmet kan fortsette.
278 – Adressen til *connect_work* blir lagt i workqueue.
33 – Denne vil sjekke om utviklingskortet er koblet til skytjenesten, denne funksjonen vil gå i loop til oppkoblingen er vellykket.
85 – *cloud_event_handler* sjekker status til skyoppkobling.
103 – Oppkoblingen er vellykket og *connect_work* slettes fra workqueue (linje 33).
238 – Denne blir trigget ved knappetrykk på *button1.*
241 – Legger adressen til *cloud_update_work* til workqueue.
56 – Gjør klart for sending av data, *CONFIG_CLOUD_MESSAGE* er definert i Autoconfig.h ***
67 – Sjekker at den definerte skytjenesten er "NRF_CLOUD".
73 – *cloud_send* sender data til skyen, parameter nr2 er adressen til meldingsbufferet.
Tilbake til 238 og venter på nytt knappetrykk.

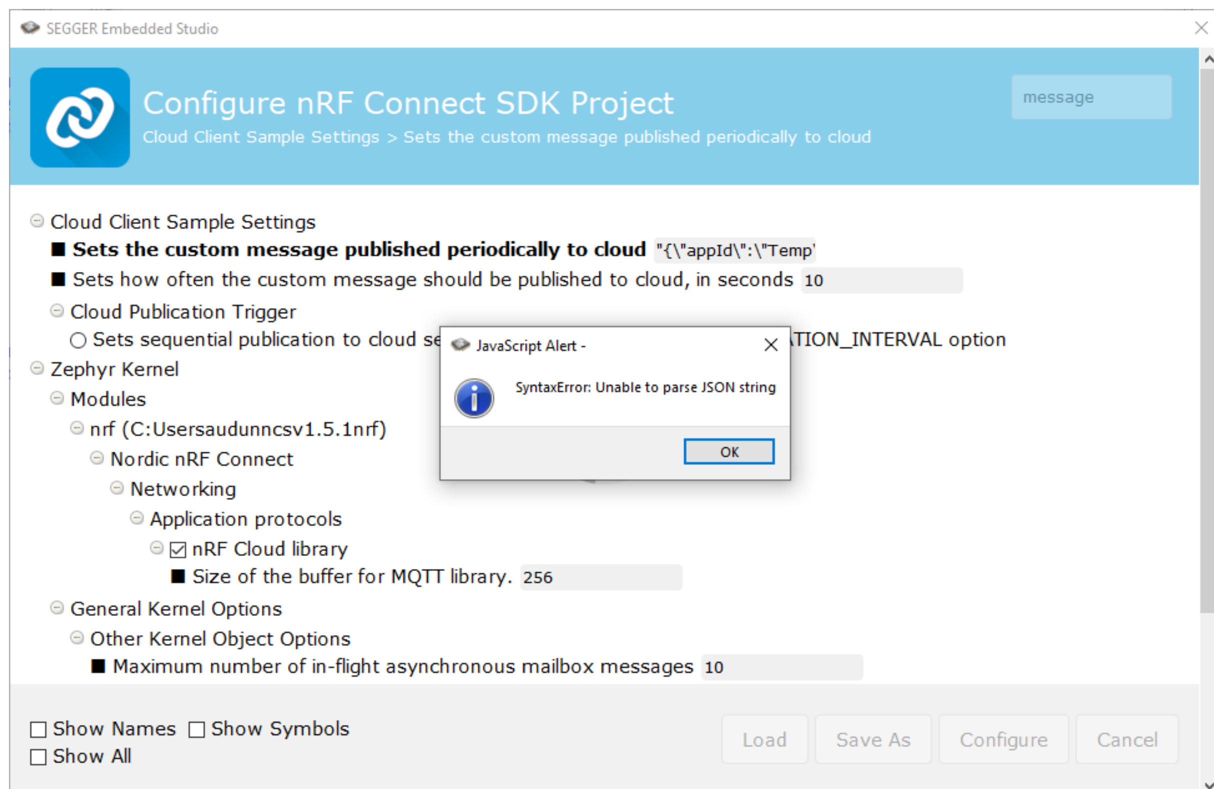
*** Dette er strukten som blir brukt til sending av data. Den består av QoS (Quality of Service), en melding og lengden av meldingen. Data blir sendt til skyen når funksjonen *cloud_send* blir kjørt. *CONFIG_CLOUD_MESSAGE* settes opp i konfigurasjonsverktøyet og er en streng på JSON format.

Hvis vi ser på teksten fra konfigureringsverktøyet sammenliknet med den fra Autoconf.h ser vi at det er lagt til ' \ ' før alle ' " '. Dette er for å fortelle kompilatoren at neste tegn ikke er avslutningen på tekststrengen.

Konfigurasjonsverktøy: {"state":{"reported":{"message":"Hello Internet of Things!"}}}
Autoconf.h: "{\"state\":{\"reported\":{\"message\":\"Hello Internet of Things!\"}}}"

Jeg eksperimenterte med egne tekststrenger for å se om jeg kunne få sendt temperaturdata til riktig vindu i nRF Cloud, underveis kom jeg i skade for å legge inn følgende i konfigurasjonsverktøyet.
"{\"appId\":\"Temp\",\"data\":\"21\"}"

Dette resulterte i feilmeldingen vist i figur 19, eneste valg er "OK" og dette gjør at systemet låser seg. Man kan da ikke lengre åpne verktøyet fordi syntaksen er feil… I Autoconf.h har vi nå strengen.
#define CONFIG_CLOUD_MESSAGE "\"{\\\"appId\\\":\\\"Temp\\\",\\\"data\\\":\\\"21\\\"}\""
Vakkert, ikke sant?



Figur 19 – CONFIG_ CLOUD _MESSAGE

Så hvordan fungerer "*Configure nRF Connect SDK Project*" og hvordan er koblingen til Autoconf.h. Hvordan får man dette i gang igjen? Løsningen skulle bli å bruke WinMerge for å markere filer som var forandret fra det opprinnelige prosjektet. Før jeg kom frem til denne løsningen ble det forsøkt å benytte søkefunksjonen i Windows for å finne "*CONFIG_CLOUD_MESSAGE*", men det skulle vise seg at Windows hverken søker i filer uten "extensions" eller filer uten navn. Disse filene er relevante:

- (USERNAME) \ncs\v1.5.1\nrf\samples\nrf9160\cloud_client – Kconfig
  - Denne blir brukt for innledende konfigurasjon og oppsett av konfigurasjonsverktøyet.
- (USERNAME) \ncs\v1.5.1\nrf\samples\nrf9160\cloud_client\build_nrf91(..)\zephyr – .config
  - Når konfigurasjonsverktøyet åpnes blir konfigureringen lastet inn fra denne fila. Det er denne fila som blir brukt for å generere Autoconfig.h
- (USERNAME) \ncs\v1.5.1\nrf\samples\nrf9160\cloud_client\build_nrf9160dk_nrf9160ns \zephyr\include\generated – Autoconfig.h
  - Dette er resultatet av kompileringen.

Løsningen på problemet var å endre strengen i .config med følgende.
CONFIG_CLOUD_MESSAGE = "{\"appId\":\"Temp\",\"data\":\"21\"}"

5. Kode – motta data fra sky

I mange tilfeller er det praktisk å kunne gi kommandoer eller beskjeder til et system som er koblet på nettet. For eksempel vil du ha en statusoppdatering eller kanskje du vil styre en lampe. Vi skal i dette kapittelet se på hvordan man kan sende en melding fra nRF Cloud og tolke den på kortet. Som nevnt tidligere kreves det at man sender meldinger på JSON format fra nRF Cloud.

Vi så i figur 18 på **cloud_event_handler** som er en callback-rutine fra *lte_handler*. Denne blir kjørt hver gang det skjer en hendelse i cloud-backend. Dersom vi mottar en innkommende melding vil den tilhørende hendelsen være "*CLOUD_EVT_DATA_RECEIVED*". Meldingen blir da skrevet ut til terminal. For å tolke denne meldingen og utføre en handling trenger vi en JSON parser. Min far, Tord Brabrand, har jobbet med noe tilsvarende tidligere og hadde en enkel parser som kunne tilpasses til dette bruksområdet. Han har bistått meg med tilpasning og implementasjon i denne delen av oppgaven. Jeg kommer ikke til å gå i detalj på hvordan parseren er bygget opp da det ikke er direkte relevant til etterarbeid for denne rapporten. Man vil fort ha behov for en mer universell parser som kan ta alle typer meldinger. (47)

Målet med denne delen av oppgaven vil være å kunne kontrollere de fire lysdiodene på kortet samt innhente sensordata. Som nevnt tidligere har ikke utviklingskortene til Nordic innebygde sensorer, så vi vil derfor benytte simulerte sensorverdier for denne demoen. Driverbiblioteket til Nordic inneholder en simulator som kan gi simulerte verdier til de funksjonskallene man vanligvis henter sensordata fra, dersom det aktuelle kortet ikke har sensorene man forsøker å benytte. JSON parseren er tilpasset for å fungere til dette bruksområdet. Kildefiler til dette prosjektet ligger som vedlegg til rapporten, se "*cloud_client_data_received.zip*". For å komme i gang med dette prosjektet kopierer man inn filene til mappen

- (USERNAME) \ncs\v1.5.1\nrf\samples\nrf9160\cloud_client\src

Så må vi fortelle kompilatoren at vi ønsker å ha med *json_command_interpreter.c og json_command_interpreter.h*, dette gjøres ved å legge dem til i *CMakeList.txt,* ett nivå opp fra der du la inn prosjektfilene. I denne fila legger du inn de nye filene på samme måte som main.c er lagt inn. Når dette er gjort må CMake kjøres og prosjektet vil være kompilerbart.

For at prosjektet skal fungere må man inn i konfigureringsverktøyet for å aktivere *Sensor Driver* (søk etter *sensor*). Trykk konfigurer. Så må vi legge til *Sensor simulator*, dette vil dukke opp etter at konfigureringen av *Sensor Drivers* er kjørt. Etter dette må du aktivere *Sensor simulator trigger*. Du må altså inn i konfigurasjonsverktøyet tre ganger for å konfigurere dette.

Jeg vil nå begynne å snakke om selve koden i prosjektet. For å ikke gjøre dette alt for omfattende vil ikke absolutt alt bli forklart her. Tanken er at kommentarene underveis i koden er forklarende på egenhånd og at jeg derfor konsentrerer meg om å forklare tankegangen bak oppbygningen. Denne koden er en utvidelse av koden i forrige kapittel, jeg vil derfor fortsette der jeg slapp og nå forklare mottak av data og til en viss grad hvordan dataene blir tolket.

cloud_event_handler vil bli kjørt når systemet mottar en melding fra skyen og vil i sin tur kjøre **case** CLOUD_EVT_DATA_RECEIVED som frigir semaforen *json_received*. Koden vist i figur 20 vil bli kjørt når semaforen er frigitt. **k_sem_take** vil suspendere main-løkka inntil semaforen er tilgjengelig.

```
435         while (1)  // Main-loop to handle received json command strings:
436         {
437                 k_sem_take(&json_received, K_FOREVER);
438 
439                 LOG_INF("JSON message received: %s\n", rx_buffer);
440                 ParseJsonString(rx_buffer);
441                 HandleUpdatedAttibuteValues();
442 
443                 if (get_command_received)  // If a "get" command is received ..
444                 {
445                     SendAttributeStatusToCloud();  // the requested value will be send to cloud
446                     get_command_received = FALSE;
447                 }
448         }
```

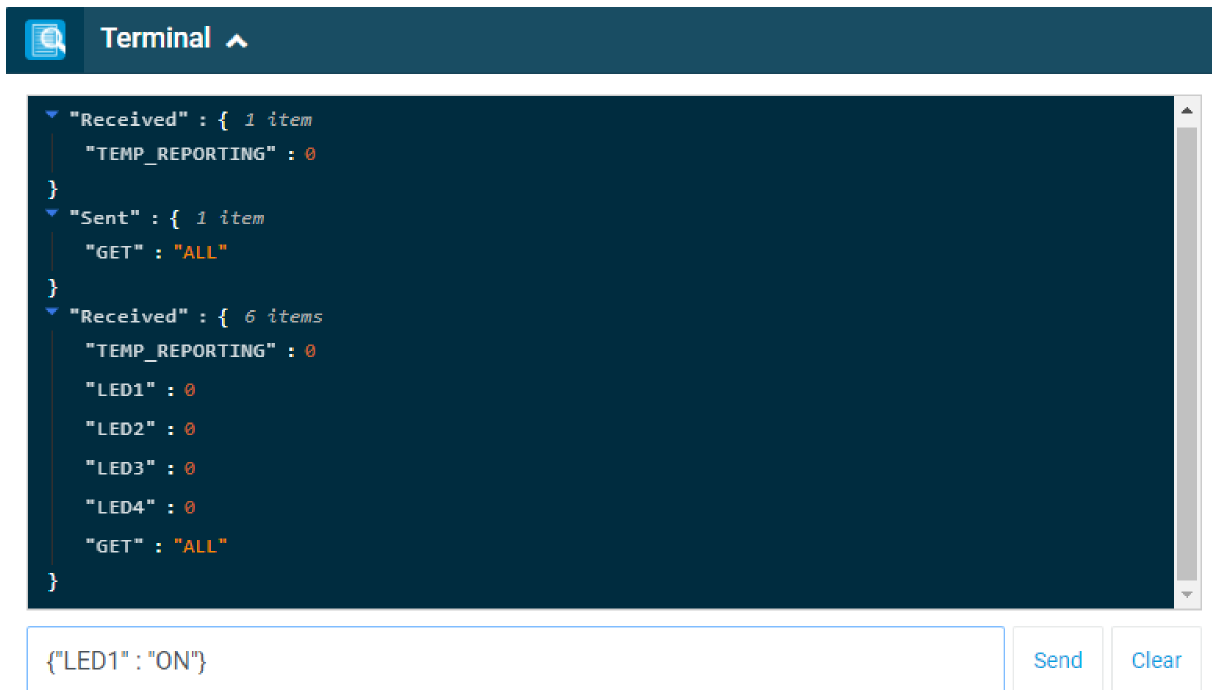
Figur 20 – JSON mottatt

Dette vil så starte funksjonen **ParseJsonString** som plukker ut teksten eller ordene fra JSON meldingen. Denne vil sortere ut hva som er *topic* og hva som er nytteverdien (payload) i hver melding. Dette blir gjort i den eksterne fila *json_command_interpreter.c*. Når dette er kjørt vil programmet fortsette med funksjonen **HandleUpdatedAttributeValues** som går gjennom de ordene som forrige funksjon plukket ut og sjekke disse opp mot en liste med forhåndsdefinerte attributter. Denne vil igjen kalle funksjonen **HandleNewValue** som gjør handlinger basert de data parseren har behandlet. Etter dette vil vi komme tilbake til main-løkka. Hvis *get_command_received* er forskjellig fra NULL vil funksjonen **SendAttributeStatusToCloud** bli kjørt.

Jeg har forhåndsdefinert følgende attributter i systemet (figur 21), disse kan slås av eller på ved å sende "ON" eller "OFF" samt 0 eller 1 til systemet. Ved å skrive {"GET" : "ALL"} kan man få status på alle attributter i systemet. Parseren var lagt opp til å ikke være case-sensitiv, dermed har det ikke noe å si hvordan du skriver meldingene til dette systemet. *TEMP_REPORTING* styres på samme måte som lysdiodene, når denne er aktivert vil det sendes simulerte temperaturdata til skyen hvert 5 sekund. Figur 22, 23 og 24.

```
43     Attribute table:
44 */
45 #define a_TEMP_REPORTING            0
46 #define a_LED1                      1
47 #define a_LED2                      2
48 #define a_LED3                      3
49 #define a_LED4                      4
50 #define a_GET                       5
```

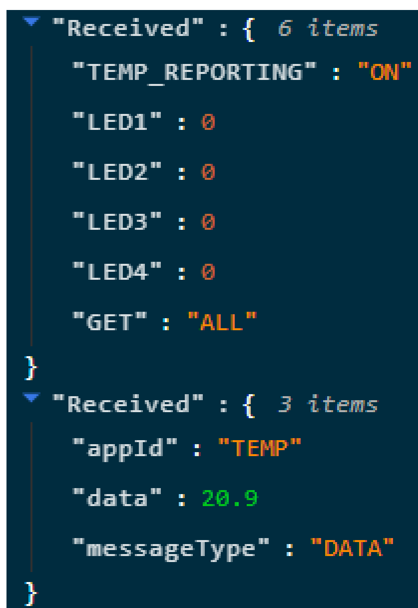
Figur 21 – Attributter



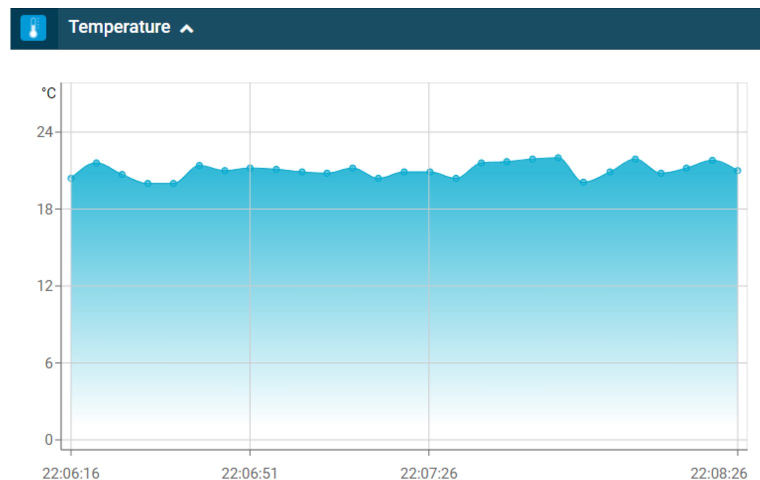
Figur 22 – nRF Cloud

5.1 Sensor avlesning

Det siste vi skal ta for oss er hvordan man leser av en sensor. Vi starter med å få dette opp å gå i prosjektet *minimal*. Det er ikke nødvendig at du har noe tidligere konfigurering i dette prosjektet utenfor den vanlige konfigureringen prosjektet har når det lastes. Nok en gang kreves det at vi gjør en del konfigurering. *Sensor simulator trigger* må aktiveres, dette innebærer at du må inn i konfigureringsverktøyet "*Project*" -> "*Configure nRF Connect SDK Project...*" tre ganger. Søk etter *sensor* for å få opp valgene. I tillegg må *Newlib C library –> Build with newlib-nano C library* konfigureres. Figur 25 viser koden for sensoravlesning, data fra sensoravlesningen blir skrevet ut til terminal.



Figur 24 – TEMP_REPORTING



Figur 23 – Temperatur

```c
/* ----- Read sensor data ----- */
#include <zephyr.h>
#include <drivers/sensor.h>

void main(void)
{
    int err;
    const struct device *dev;
    struct sensor_value data;
    double value = 123.0;
    struct k_spinlock lock;
    k_spinlock_key_t key;

    dev = device_get_binding(CONFIG_SENSOR_SIM_DEV_NAME);
    __ASSERT(dev, "Could not get device %s\n", CONFIG_SENSOR_SIM_DEV_NAME);

    while (1)
    {         //Fetching sensor value from channel to internal buffer
        err = sensor_sample_fetch_chan(dev, SENSOR_CHAN_AMBIENT_TEMP);
        if (err)
            printk("Failed to fetch data from %s, error: %d", CONFIG_SENSOR_SIM_DEV_NAME, err);
              //Collecting data from internal buffer
        err = sensor_channel_get(dev, SENSOR_CHAN_AMBIENT_TEMP, &data);
        if (err)
            printk("Failed to fetch data from %s, error: %d", CONFIG_SENSOR_SIM_DEV_NAME, err);
        else
        {
            key = k_spin_lock(&lock);  //Ensure that this thread is not interrupted ...
            value = sensor_value_to_double(&data);
            k_spin_unlock(&lock, key); // ... until here
        }

        printk("Temperature = %d\n", (int)value);
        k_sleep(K_SECONDS(5));
    }
}

/* ----- Configuration ----- */
//Sensor Driver - Sensor simulator - Sensor simulator trigger
//Newlib C library - Build with newlib-nano C library
```

Figur 25 – Sensoravlesning

Første del av koden er tilsvarende til ting vi har sett på før, men så kommer vi til *Spinlock*. Spinlock brukes for å sikre at et kodeavsnitt ikke blir avbrutt. *k_spinlock_key_t* er en strukt som er definert i en ekstern fil. Ved se på "*Go to definition" ser* vi at den finnes i *spinlock.h* og er en **typedef strukt** z_spinlock_key k_spinlock_key_t;

Funksjonen **sensor_sample_fetch_chan** brukes for å lese ut sensorverdier og lagre disse til et internt buffer. Videre vil **sensor_channel_get** gi oss verdien fra dette bufret for den sensoren vi forespør. Dette bufret er todelt med verdier for tallet foran og bak komma. Man benytter funksjonen **sensor_value_to_double** for å konvertere disse til et dobbelt presisjon "floating point" tall. Vi ser i eksempelet her må man være sikker på at verdien i *data* ikke blir endret underveis i prosessen med å slå dem sammen. Det er derfor viktig å benytte funksjoner som *spinlock*.

Det er denne koden som er implementert i eksempelet med skytilkobling. *Temp_reporting* skrur av eller på den rutinen som henter sensordataene. Funksjonen **send_temperature_thread** kjøres av en thread og vil etter initialisering kjøres konstant. **while**(1) vil alltid gå i loop med intervall på TEMP_REPORTING_INTERVAL. Dersom *temp_reporting* er satt vil rutinen kjøre kode tilsvarende den i figur 25. Den vil til slutt kalle **MakeJsonTemperatureString** som tar temperaturen og pakker den inn i en JSON streng som tolkes til temperatur i nRF Cloud.

Når koden startes og LTE forbindelsen er opprettet vil LED1 tennes, LED2 tenner når oppkoblingen til skytjenesten er vellykket. Det gjør det enkelt å følge prosessen underveis. Figur 24 og figur 25 viser resultatet når *TEMP_REPORTING* er satt til 1 (ON). Funksjonen **SendTemperatureToCloud** kaller **cloud_send** som gjør selve publiseringen av meldingen.

5.2 Oppsummering

Dette er i bunn og grunn alt du trenger å vite for å komme i gang med IoT på utviklingskort fra Nordic Semiconductor som kjører Zephyr RTOS. Vi har sett på hvordan de grunnleggende prinsippene med Threads og Workqueue fungerer, samt hvordan en lysdiode styres. Vi har fått etablert kobling til skyen og sett på hvordan kommunikasjonsflyten går. Vi har også vært inne på hvordan kortet kan kontrolleres via skyen. Ved å lese mer om *button_init* finner man ut av hvordan funksjoner trigges fra knappetrykk. Selv om det ikke er noen perfekt kode kan JSON parseren utvides for bruk i flere bruksområder.

Vi har også vært innom konfigurering og sett på hvordan man benytter konfigureringsverktøyet for å aktivere ytterlige funksjoner i koden. Det har blitt presentert en del nyttige artikler som forklarer temaer det kan være lurt å lese mer om. Jeg vil avslutningsvis komme med noen konkrete ting man kan se videre på.

Vi har dessverre ikke brukt så mye tid til å snakke om Bluetooth, noe av grunnen til dette er at det er et stort fagfelt i seg selv. I tillegg har nRF9160 blitt brukt for å demonstrere prosjektet, denne mikrobrikken har som sagt ikke støtte for Bluetooth. Dersom vi skulle fått i gang et Bluetooth-prosjekt måtte vi fått til en kobling mellom nRF52840 og nRF9160 på utviklingskortet.

Dessverre har jeg ikke lyktes i prosessen med å migrere prosjektet over til å kjøres på Thingy:91. Jeg har forsøkt å konfigurere prosjektet etter dette tipset for å få generert *app_signed.hex*:
https://devzone.nordicsemi.com/f/nordic-q-a/58024/no-app_signed-hex-in-build-folder
Og jeg har fulgt denne veiledningen for å klargjøre og laste kode på Thingy:91:
https://infocenter.nordicsemi.com/index.jsp?topic=%2Fug_nc_programmer%2FUG%2Fnrf_connect_programmer%2Fncp_programming_thingy91.html
Til tross for dette får jeg feilmeldingen "*MCUboot DFU failed*" når prosjektet forsøktes lastet til kortet. Jeg har derimot lyktes med å laste "*asset_tracker*"-prosjektet på denne måten. Det kan virke som konfigureringen er feil i "*Cloud_client*" for å tillate programmering på denne måten. (48) (49)

6. Etterarbeid

Vi har nå satt opp et system med få muligheter for utvidelser. nRF Cloud er ingen ideell løsning for et generelt IoT sensornettverk. Vi har også kun simulerte sensorer som gir oss unyttige verdier i et reelt system. Derfor ville det være spennende å lære mer om AWS IoT (Amazon Web Services) og ta i bruk deres løsninger for MQTT broker og web applikasjoner. Node-RED er et grafisk konfigureringsverktøy man kan bruke for oppsett av sensornettverk. (50) (51) (52)

JSON parseren som ble presentert i forbindelse med denne rapporten er også noe man må se videre på. Det finnes en JSON parser i *asset_tracker* prosjektet, men førsteinntrykket var at det var over mitt ambisjonsnivå å finne ut av på den tiden jeg hadde til rådighet. Jeg har heller ikke brukt veldig mye tid på å Google etter andres løsninger på slike parsere, så her gir jeg fra meg stafettpinnen uten videre tips.

Videre ville det vært artig å utvidet systemet med Bluetooth for å ha flere sensorer over et område som kan jobbe sammen i nettverket. Nordic har lagt opp til at man kan benytte Bluetooth MESH fra BLE standarden, og med dette kan man ha lang avstand mellom den ytterste noden i systemet og sentralen. Dette kan for eksempel benyttes i jordbrukssammenheng da man kan plassere flere sensorer ut på et jorde, og som kan transportere data over større avstander. Bluetooth MESH bruker mer strøm enn en vanlig BLE tilkobling, derfor bør man se på mulighetene for å ha noen noder med større batteri/energihøsting som kan være mellommenn, mens sensorene ellers bare sender et pling når de har noe å melde. (53)

Jeg har heller ikke gått opp veien for å finne ut hvordan man konfigurerer opp et nytt tomt prosjekt i SEGGER. Det ville vært gøy å lagt ut et eget kretskort med mikrobrikkene fra Nordic, og startet fra grunn av med å sette opp all nødvendig konfigurering. Hvor omfattende dette er vet jeg ikke, det kan være nyttig å ta for seg disse to trådene fra DevZone for å ha et sted å starte.

- Devicetree konfigurering:
  - https://devzone.nordicsemi.com/f/nordic-q-a/47990/device-tree-configuration-guidelines
  - https://devzone.nordicsemi.com/nordic/nrf-connect-sdk-guides/b/peripherals/posts/adding-a-peripheral-to-an-ncs-zephyr-project
- Amazon Web Services
  - https://aws.amazon.com/iot/
  - https://www.youtube.com/watch?v=T_Cfepvd2hw
- Mosquitto MQTT
  - http://www.steves-internet-guide.com/mosquitto_pub-sub-clients/
- Node-RED
  - https://nodered.org/
  - https://www.youtube.com/watch?app=desktop&v=GeN7g4bdHiM
- Bluetooth Low Energy
  - https://www.bluetooth.com/bluetooth-resources/intro-to-bluetooth-gap-gatt/
  - https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt
- Bluetooth prosjekt
  - https://circuitdigest.com/microcontroller-projects/nordic-nrf52-development-kit-measuring-temperature-and-humidity-using-ble
- Koble til LCD display
  - https://devzone.nordicsemi.com/nordic/nrf-connect-sdk-guides/b/peripherals/posts/lvgl-on-a-tft-lcd-display-with-the-nrf9160-dk

7. Refleksjon

Dette har vært et utrolig spennende og lærerikt prosjekt som har gitt meg en god innføring i IoT og sanntidsprogrammering. Det er ingen tvil om at det var tungt å komme i gang med systemene, men etter en strevsom start begynte ting å falle på plass og prosessen ble mer og mer morsom etter hvert som tiden gikk. Jeg ble advart på forhånd om at det var en forholdsvis bratt læringskurve for dette systemet, men at når det først løsnet ville det gå fort videre. Det var litt skuffende å se hvor få nybegynner-guider det finnes innenfor dette tema, men jeg håper at denne rapporten vil være til hjelp for andre som ønsker å komme i gang.

Det må sies at det har vært en god del frustrasjon underveis. Det er slettes ikke alltid like lett å skaffe oversikt over de temaene man prøver å sette seg inn i, og når man først begynner å grave er det bare toppen av isfjellet man har funnet. I moderne embedded programmering blir det vanskeligere og vanskeligere å lage systemer fra bunnen av. Man får derfor aldri fullstendig oversikt og man må bare stole på at de som har utviklet driverne har gjort en god jobb.

Konfigurasjon er noe jeg har begynt å få oversikt over, men det er fortsatt usikkerhetsmomenter knyttet til dette. Dessuten er hele systemet med Devicetree og hvordan man skulle definert pinout for et eget kretskortutlegg noe som det gjenstår å finne ut av.

Potensialet for denne typen teknologi, og de mikrobrikker som Nordic har designet gjør at det uten tvil er verdt å bruke tid på dette. Jeg er fristet til å si at det kun er fantasien som setter grenser for hva man kan utvikle når man har kontroll og oversikt over et slikt system.

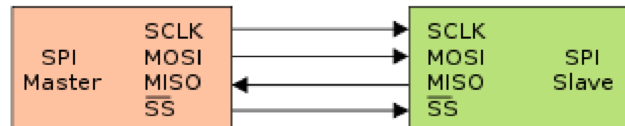
ESP32 er et konkurrerende system, men som foreløpig ikke tilbyr LTE eller GPS teknologier. Ellers er dette mye brukt i DIY (gjør det selv) sammenheng, så tilgangen på guider og eksempelprosjekter er mye bedre. Det kan være verdt å lese mer om dette for systemer der strømforbruk og mangel på LTE ikke er så avgjørende. (54)

Dette er absolutt et tema jeg vi anbefale andre å sette seg inn i, og jeg håper at jeg med denne rapporten har gjort det litt mer overkommelig.

8. Kjekt å vite

8.1 SPI og I2C

SPI og I2C er kommunikasjonsprotokoller brukt for kommunikasjon mellom en master og opptil flere slaver. Det kan typisk være mellom en microkontroller og andre integrerte kretser. Felles for begge protokollene er at det kreves felles klokke og jord for å overføre data. SPI baserer seg på "full-duplex" med 2 datalinjer i motsetning til "halv-duplex" på I2C der kommunikasjonen bare går over en datalinje. Se figurer. I SPI har man SS (Slave Select) som forteller hvilken slave master ønsker å snakke med, masteren må ha en SS pinne for hver slave som skal kobles til. I2C på sin side starter hver sending med adressen til den slaven som man ønsker å kommunisere med. SPI støtter også langt høyere båndbredde enn det I2C støtter, gjerne over 10megabit/sekund. (55) (56)



Figur 26 – SPI

8.2 UART

Universal asynchronous receiver-transmitter (UART) er hardware for asynkron seriell kommunikasjon mellom to enheter. Typisk kan det være mellom en microkontroller og en sensor, mellom to microkontrollere eller mellom en datamaskin og en microkontroller. RS-232 (12v) og RS485 (5v) er mye brukte i UART sammenheng. (57)

8.3 Nettverksstandard - OSI modellen

Open system interconnection (OSI) modellen beskriver hvordan de ulike protokollene i et datanettverk fungerer sammen. Nettverket er delt opp i fem lag som hver har ansvaret for sin del av kommunikasjonen. Siden et nettverk er svært komplekst har det vært praktisk å dele det opp i ulike deler, hvert lag trenger bare å forholde seg til APIet til laget under seg. (58) Oppdelingen er som følger:

5. **Applikasjonslaget**
   Programvaren som ønsker å kommunisere sammen.
   HTTP, FTP, MQTT

4. **Transportlaget**
   Pålitelig overføring av datasegmenter mellom to punkter i nettverket.
   Korrigere eventuelle feil i underliggende lag.
   TCP protokoll, UDP protokollen

3. **Nettverkslaget**
   Meldingshåndtering på et multi-node nettverk.
   Data adresseres til riktig mottaker i riktig rekkefølge.
   IP protokollen

2. **Datalinklaget**
   Overføring av data mellom to noder på det fysiske laget.
   Fysisk adressering, (MAC adresser).
   Ethernet protokoll

1. **Fysisk lag**
   Overføring av bitstrøm over et fysisk medium.
   Elektriske kabler, fiberoptikk og radiobølger

9. Referanser

1. Telenor. Internet of Things (IoT). [Online].; 2020 [cited 2020 desember. Available from: https://www.telenor.no/bedrift/iot/.

2. Telenor. IoT/M2M. [Online].; 2020 [cited 2021 Februar. Available from: https://bedriftsbutikk.telenor.no/m2m-produkter/start-iot-nordic-thingy91-telenor/.

3. Brabrand A. Forstudie - Prosjektrapport. ; 2021.

4. NTNU. Prosjektmanual for emnet TELE3001/3021/3031 Bacheloroppgave Elektro. ; 2021.

5. Gillis AS. IoT Agenda. [Online].; 2020 [cited 2021 Juli. Available from: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT.

6. Wikipedia. Nordic Semiconductor. [Online].; 2021 [cited 2021 Juli. Available from: https://en.wikipedia.org/wiki/Nordic_Semiconductor.

7. Wikipedia. Nordic Semiconductor (norge). [Online].; 2021 [cited 2021 Juli. Available from: https://no.wikipedia.org/wiki/Nordic_Semiconductor.

8. Nordic Semiconductor. Thingy:52. [Online].; 2021 [cited 2021 Mai. Available from: https://www.nordicsemi.com/Products/Development-hardware/Nordic-Thingy-52.

9. Nordic Semiconductor. Thingy:91. [Online].; 2021 [cited 2021 Mai. Available from: https://www.nordicsemi.com/Products/Development-hardware/Nordic-Thingy-91.

10. Nordic Semiconductor. nRF9160. [Online].; 2021 [cited 2021 Januar. Available from: https://www.nordicsemi.com/Products/nRF9160.

11. Nordic Semiconductor. nRF9160 Product Brief. [Online]. [cited 2021. Available from: https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF9160-SiP-product-brief.pdf?sc_trk=&la=en&hash=5C889507F72CE933BE712F9748D6FE19600C6746.

12. ARM Developer. ARM Cortex-M series processors. [Online].; 2021 [cited 2021 Juli. Available from: https://developer.arm.com/ip-products/processors/cortex-m.

13. Nordic Semiconductor. nRF52840 SoC. [Online].; 2021 [cited 2021. Available from: https://www.nordicsemi.com/Products/nrf52840.

14. Nordic Semiconductor. nRF52832 SoC. [Online].; 2021 [cited 2021. Available from: https://www.nordicsemi.com/Products/nrf52832.

15. SEGGER Embedded Studio. SEGGER Embedded Studio. [Online].; 2021 [cited 2021 Mai. Available from: https://www.segger.com/products/development-tools/embedded-studio/.

16. Nordic Semiconductor. nRF Connect. [Online].; 2021 [cited 2021 Januar. Available from: https://www.nordicsemi.com/Products/Development-tools/nrf-connect-for-desktop.

17. Kirsch CM. Principles of Real-Time Programming. [Online].; 2002 [cited 2021 Juni. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.8.9760&rep=rep1&type=pdf.

18. Zephyr Project. Benefits with Zephyr. [Online].; 2021 [cited 2021 Februar. Available from: https://www.zephyrproject.org/benefits/.

19. Wikipedia. Zephyr (operating system). [Online].; 2021 [cited 2021 Juli. Available from: https://en.wikipedia.org/wiki/Zephyr_(operating_system).

20. Wikipedia. Zephyr (operating system). [Online].; 2021 [cited 2021 Mai. Available from: https://en.wikipedia.org/wiki/Zephyr_(operating_system).

21. Nordic Semiconductor. Working with nRF9160 DK. [Online].; 2021 [cited 2021 Mars. Available from: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/ug_nrf9160.html#available-drivers-libraries-and-samples.

22. Zephyr Project. System Threads. [Online].; 2021 [cited 2021 Juli. Available from: https://docs.zephyrproject.org/1.9.0/kernel/threads/system_threads.html.

23. Zephyr Project. Workqueue Threads. [Online].; 2021 [cited 2021 Mars. Available from: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/reference/kernel/threads/workqueue.html.

24. Zephyr Project. Kernel APIs. [Online].; 2021 [cited 2021 Mars. Available from: https://docs.zephyrproject.org/1.9.0/api/kernel_api.html?highlight=k_delayed_work_init#_CPPv319k_delayed_work_initP14k_delayed_work16k_work_handler_t.

25. Zephyr Project. k_work. [Online].; 2021 [cited 2021 Juli. Available from: https://docs.zephyrproject.org/apidoc/latest/structk__work.html.

26. Zephyr Project. k_work_q. [Online].; 2021 [cited 2021 juli. Available from: https://docs.zephyrproject.org/apidoc/latest/structk__work__q.html.

27. Zephyr Project. Device Driver Model. [Online].; 2021 [cited 2021 Juni. Available from: https://docs.zephyrproject.org/latest/reference/drivers/index.html.

28. Zephyr. Devicetree Guide. [Online].; 2021 [cited 2021 Juni. Available from: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/guides/dts/index.html.

29. Android. Device Tree Overlays. [Online].; 2021 [cited 2021 Juni. Available from: https://source.android.com/devices/architecture/dto.

30. Nordic Semiconductor. nRF9160 DK Hardware. [Online].; 2020 [cited 2021 Mai. Available from: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fug_nrf91_dk%2FUG%2Fnrf91_DK%2Fmcu_virtual_com_port.html.

31. Wikipedia. NB-IoT. [Online].; 2021 [cited 2021 Juli. Available from: https://en.wikipedia.org/wiki/Narrowband_IoT.

32. Nordic Semiconductor. nRF9160 Product Specification. [Online].; 2020 [cited 2021 Januar. Available from: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fps_nrf9160%2Fproduct_overview.html.

33. Faludi R. What are the Differences Between LTE-M and NB-IoT Cellular Protocols. [Online].; 2017 [cited 2020 Desember. Available from: https://www.digi.com/videos/what-are-the-differences-between-lte-m-and-nb-iot.

34. GSM Association. NB-IoT Deployment Guide to Basic Feature set Requirements. [Online].; 2019 [cited 2021 Januar. Available from: https://www.gsma.com/iot/wp-content/uploads/2019/07/201906-GSMA-NB-IoT-Deployment-Guide-v3.pdf.

35. Nordic Semiconductor. Asset Tracker. [Online].; 2021 [cited 2021 Januar. Available from: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/applications/asset_tracker/README.html.

36. Amazon. What is AWS IoT. [Online].; 2021 [cited 2021 Juni. Available from: https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html.

37. Nordic Semiconductor. AWS IoT (Nordic). [Online].; 2021 [cited 2021 Juni. Available from: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/include/net/aws_iot.html.

38. Wikipedia. MQTT. [Online].; 2021 [cited 2021 Februar. Available from: https://en.wikipedia.org/wiki/MQTT.

39. Steve's Internet Guide. Introduction to MQTT-SN (MQTT for Sensor Networks). [Online].; 2019 [cited 2021 Februar. Available from: http://www.steves-internet-guide.com/mqtt-sn/.

40. Amazon Web Service. MQTT Communication Patterns. [Online].; 2021 [cited 2021 Februar. Available from: https://docs.aws.amazon.com/whitepapers/latest/designing-mqtt-topics-aws-iot-core/mqtt-communication-patterns.html.

41. AssetWolf. Quality of Service (QoS). [Online].; 2020 [cited 2021 April. Available from: https://assetwolf.com/learn/mqtt-qos-understanding-quality-of-service.

42. HIVEMQ. Last Will and Testament - MQTT Essentials: Part 9. [Online].; 2015 [cited 2021 April. Available from: https://www.hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament/.

43. Nordic Semiconductor. nRF9160 serial lte modem flash memory issue. [Online].; 2021 [cited 2021 Mars. Available from: https://devzone.nordicsemi.com/f/nordic-q-a/72658/thingy91-serial-lte-modem-flash-memory-issue.

44. Zephyr Project. Logging. [Online].; 2021 [cited 2021 Juli. Available from: https://docs.zephyrproject.org/latest/reference/logging/index.html?highlight=log_inf#c.LOG_INF.

45. Zephyr Project. ASSERT. [Online].; 2018 [cited 2021 Juni. Available from: https://docs.zephyrproject.org/1.9.0/reference/kconfig/CONFIG_ASSERT.html?highlight=__assert.

46. Zephyr Project. k_delayed_work_init. [Online].; 2018 [cited 2021 Juni. Available from: https://docs.zephyrproject.org/1.9.0/api/kernel_api.html?highlight=k_delayed_work_init#_CPPv319k_delayed_work_initP14k_delayed_work16k_work_handler_t.

47. Brabrand T. Veiledning med JSON parser. 2021. Kode for og hjelp med implementasjon av JSON parser i prosjektet.

48. Nordic Semiconductor. Building and programming from source code. [Online].; 2021 [cited 2021 Juli. Available from: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/1.6.1/nrf/ug_thingy91.html.

49. Nordic Semiconductor. Programming applications through USB (MCUBoot). [Online].; 2021 [cited 2021 Juli. Available from: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fug_nc_programmer%2FUG%2Fnrf_connect_programmer%2Fncp_programming_appln_thingy91.html.

50. Nordic Semiconductor. AWS - Getting Started. [Online].; 2021 [cited 2021 Mai. Available from: https://nordicsemiconductor.github.io/asset-tracker-cloud-docs/v1.5.x/docs/aws/GettingStarted/Index.html.

51. Nordic Semiconductor. Asset Tracker v2. [Online].; 2021 [cited 2021 Mai. Available from: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/applications/asset_tracker_v2/README.html.

52. Node-RED. Node-RED. [Online].; 2021 [cited 2021 januar. Available from: https://nodered.org/.

53. Eastern Peak. IoT in Agriculture. [Online].; 2020 [cited 2021 Juli. Available from: https://easternpeak.com/blog/iot-in-agriculture-technology-use-cases-for-smart-farming-and-challenges-to-consider/.

54. ESPRESSIF. ESP32. [Online].; 2021 [cited 2021 Januar. Available from: https://www.espressif.com/en/products/socs/esp32.

55. Wikipedia. Serial Peripjeral Interface. [Online].; 2021 [cited 2021 Mars. Available from: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface.

56. Wikipedia. Inter-Integrated Circuit. [Online].; 2021 [cited 2021 Mars. Available from: https://en.wikipedia.org/wiki/I%C2%B2C.

57. Wikipedia. Universal asynchronous receiver-transmitter. [Online].; 2021 [cited 2021 Juli. Available from: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter.

58. Microship. TCP/IP Five-Layer Software Model Overview. [Online].; 2021 [cited 2021 April. Available from: https://microchipdeveloper.com/tcpip:tcp-ip-five-layer-model.