NTNU
Det skapende universitet

Embedded utvikling på en fjernstyrt kartleggingsrobot

Kristian Lien

Master i kybernetikk og robotikk
Innlevert: juni 2017
Hovedveileder: Tor Engebret Onshus, ITK

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

Embedded utvikling på en fjernstyrt kartleggingsrobot

Masteroppgave

Vår 2017

Kristian Lien

Institutt for teknisk kybernetikk

Norges teknisk-naturvitenskapelige universitet

NTNU

Oppgavetekst

Det er ønskelig å bruke roboter til å kartlegge et ukjent område. Et system som gjør dette har vært under utvikling av studenter ved NTNU i flere år. Dette arbeidet har nådd et punkt der det finnes en styringsapplikasjon og et par roboter som kontrolleres av denne.

Oppgaven dreier seg rundt én av robotene i systemet. Denne er basert på Mindstorms NXT-settet til Lego. Roboten skal videreutvikles for å fikse problemer den har, og for å få på plass komponenter den mangler som de andre roboten tar i bruk. Dette skal gjøres med målet om å gjøre robotens kartleggingen mer nøyaktig. Dette inkluderer:
- Gjøre den toveis kommunikasjonen mellom roboten den eksterne kommunikasjonsmodulen kollisjonsfri
- Øke antall avstandssensorer fra to til fire
- Implementere kompass og gyroskop for nøyaktigere posisjonering
- Endre robotens operativsystem til FreeRTOS

Oppgaven innebærer også et arbeid felles for hele systemet: å forbedre kommunikasjonsmåten mellom robotene og styringsapplikasjonen. Kommunikasjonen skjer over Bluetooth mellom dongler robotene og datamaskinen er utstyrt med. Forbedringen vil gå ut på å:
- Gjøre det mulig å sende større meldinger
- Implementere en TCP-lignende protokoll for å sikre at meldinger ikke går tapt
- Gjøre kommunikasjonen adressebasert

Forord

Denne rapporten beskriver masteroppgaven jeg har arbeidet med våren 2017 ved Institutt for teknisk kybernetikk på NTNU. Masteroppgaven har gitt meg muligheten til å et helt semester fordype meg i et arbeid relatert til mitt fagområde.

Siden min studieretning er anvendte datasystemer, har det vært naturlig for meg å rette arbeidet mot embedded software og hardware. Her har jeg hatt muligheten til å ta i bruk kunnskap jeg har opparbeidet meg gjennom studiene, samtidig som jeg har lært veldig mye mer og fått dybdekunnskap i flere aspekter ved embdedded systemer.

Dette har ikke bare vært en enkel prosess, det har vært mange sene kvelder og lange netter med feilsøking og fikling med løsningene mine. Hvor mange ganger jeg i løper av denne våren har tenkt "Hvorfor fungerer ikke dette!?"har jeg mistet tellingen over.

Til slutt har heldigvis det meste løst seg på en god måte, og mye kunnskap har blitt opparbeidet gjennom å løse disse noen ganger frustrerende problemene.

Samtidig vil jeg rette en kommentar til studentene som skal ta opp tråden der jeg og mine medstudenter nå legger ned arbeidet. Forhåpentligvis vil dere finne nytte i å lese denne rapporten. Jeg har ofte under mitt arbeid klødd meg i hodet når jeg ser på hvordan de tidligere studentene har løst ting, og sannsynligvis (uheldigvis!) kommer dere til å gjøre det samme. Da håper jeg dere viser forståelse for at her har det vært mye frem og tilbake, opp og ned, i et forsøk på å lage ny funksjonalitet som virker sammen med det andre har gjort tidligere, og det mine medstudenter har utviklet for andre deler av systemet.

Jeg vil også takke min veileder under arbeidet med masteroppgaven, Professor Tor Onshus.

Kristian Lien

Trondheim, 05.06.2017

Innhold

Forkortelser

ARQ Automatic Repeat Request

BLE Bluetooth Low Energy

BT Bluetooth

COBS Consistent Overhead Byte Stuffing

CRC Cyclic Redundancy Check

LED Light Emitting Diode

PCB Printed Circuit Board

TCP Transmission Control Protocol

UART Universal Asynchronous Receiver/Transmitter

UDP User Datagram Protocol

Kapittel 1

Oppsummering

Oppgaven startet med å få på plass en løsning for å få flere avstandssensorer tilkoblet NXT-roboten, i tilegg til gyroskop og kompass. På selve NXT er det bare fire inngangsporter der det er meningen å koble inn kontakter fra standard Lego-sensorer. Altså måtte det på plass et grensesnitt for de tredjeparts komponentene som var ønsket brukt. For å få til dette ble det bestemt at et kretskort skulle utvikles som inneholder en mikrokontroller som fungerer som et mellomledd mellom roboten og sensorene. Kretskortet skulle kobles via en ledning inn på en NXT-port, og over denne linjen kommunisere alle sensorverdiene.

Prosessen med å utvikle denne løsningen begynte med at det ble satt opp en spesifikasjon over hvilke komponenter kretskortet skulle inneholde og hvordan disse skulle kommunisere. Løsningen som ble utviklet innebar at mikrokontrolleren på kretskortet har kompass og gyroskop tilkoblet på en I2C-buss, og de analoge spenningene avstandssensorene leverer som angir målt avstand leses av en analog-digital-konverterer i mikrokontrolleren. Mikrokontrolleren ble også satt opp til å kommunisere med Bluetooth-dongelen med en UART. Det hadde tidligere vært problemer med kommunikasjonen mellom NXT og dongelen, og det ble funnet ut at dersom dongelen også blir inkludert i kretskortet kunne dette løses. En annen av UARTene på mikrokontrolleren ble satt opp til å styre en RS485-transceiver som genererer kommunikasjonssignalet mot NXT.

Denne spesifikasjonen måtte omsettes først i en prototype på et breadboard der funksjonaliteten ble testet opp mot målene, og deretter realiseres på et kretskort. For å ta steget fra prototype til et endelig kretskort ble en CAD-programvare tatt i bruk. Kretskortdesign er en to-trinns prosess, så først ble en skjematisk tegning laget som inneholder alle komponentene og de elektroniske forbindelsene mellom disse. Neste trinn var å ta i bruk denne tegningen

til å utvikle den fysiske layouten til kortet. Komponentenes fotavtrykk ble plassert på logiske steder, og de elektroniske forbindelsene fra tegningen måtte realiseres med legging av kobberbaner mellom de forskjellige inn- og utgangene. Dette trinnet var det mest tidkrevende, og bestod av mange iterasjoner med flytting av komponenter og nye forsøk på å få rutet alle kobberbanene slik at ingen krysser hverandre.

Selve kretskortet ble ikke produsert selv, men tegningene ble sendt til en produsent som gjorde dette. De forskjellige komponentene, motstander, kondensatorer, mikrokontroller og så videre, ble deretter plassert på det ferdige kretskortet og loddet fast.

Mikrokontrolleren ble satt opp til å kommunisere med NXT over en halv-duplex RS485-linje. Siden linjen er halv-duplex og begge parter derfor ikke kan sende data samtidig da det kan resultere i kollisjoner, ble det implementert en protokoll som definerer NXT som master på linjen. Mikrokontrolleren får bare lov til å sende data når NXT gir den beskjed om dette.

Mikrokontrolleren har blitt programmert til å kontinuerlig lese av sensorverdier og å motta data fra Bluetooth-dongelen. Disse blir lagret og ovserendt NXT når denne ber om dette.

Videre har et nytt operativsystem, FreeRTOS, blitt implementert på NXT. For å gjøre dette har selve NXT-boksen blitt åpnet for å blottlegge elektronikken og mikrokontrolleren inni. Her ble det loddet ledninger fast på hovedkortet som gjorde det mulig å koble til mikrokontrollerens JTAG-grensesnitt. Ledningene ble i den andre enden festet i en kontakt som ble plugget inn i et programmeringsverktøy. Med dette verktøyet var det så mulig å laste inn operativsystemet og programmer.

Drivere som implementerer grensesnitt mot de de forskjellige komponentene NXT er utstyrt med, som LCD-skjerm og RS485-modul, måtte også på plass. Slike drivere fantes i operativsystemet som hadde vært brukt tidligere, og disse ble overført og tilpasset til den nye plattformen. Når alt dette var løst ble robot-applikasjonen fra AVR-roboten, som også bruker FreeRTOS, enkelt overført til NXT.

Til slutt har det blitt utviklet et nytt kommunikasjonssystem for totalsystemet. Denne kommunikasjonen ble implementert i lag inspirert av i OSI-modellen. Et av lagene, nettverkslaget, definerer den laveste dataenheten som blir kommunisert. Denne inneholder adressen til avsender og mottaker, i tilegg til dataen som skal sendes. Ved hjelp av adressene kan donglene videresende dataen til riktig sted.

Over dette laget er det utviklet et transportlag som definerer to forskjellige kommunikasjonsprotokoller. Den ene ligner på UDP i Internets protokollkatalog, og muliggjør vanlig simpel

meldingsoverføring. Den andre er inspirert av TCP og benytter seg av automatisk retransmisjon av meldinger dersom data går tapt. Dataen disse protokollen ønsker å overføre blir gitt til nettverkslaget der de blir sendt på nettverket som en del av datafeltet i nettverks-pakken.

Kapittel 2

Summary

The first thing that was done was to implement a solution that would allow more distance sensors to be connected to the NXT, in addtion to a gyroscope and compass. The NXT itself only has four ports where standard NXT-sensors are supposed to be connected. This means that some interface to the third-party components had to be created. To achieve this it was decided to create a circuit board that contains a microcontroller that works as a middle-man between the robot and the sensors. The circuit board was to be connected via a wire to the NXT, and communicate the sensor values over this line.

The process of developing this solution started with the creation of a specification over what components the PCB should contain and how these should communicate. The solution that was created involved that the microcontroller has the gyroscope and compass connected on an I2C-bus, and the analog voltages produced by the distance sensors that vary with the distance measured are read by an analog-to-digital converter in the microcontroller. The microcontroller was also set up to communicate with the Bluetooth-dongle with UART. Previously there had been issues with the communication between the dongle and NXT, and it was suggested that including the dongle in the PCB would solve the problem. Another UART in the microcontroller was set up to control an RS485-transceiver that generates the communiation signal going to the NXT.

This specification had to be turned into a prototype on a breadboard to confirm the functionality, and then produced on a PCB. To turn the specification into a PCB design a CAD software was used. PCB-design is a two-step process that starts with the creation of a PCB-schematic that contains all the components and the electrical connections between them. The next step is to turn this schematic into a fysical PCB-layout. The component's footprints

were placed in logical positions and the electrical connections from the schematic realised using copper traces. This step was the most time consuming and involved a lot of iterations of moving components and attempts at routing all the copper traces without crossing another.

The PCB design was sent to a manufacturer that created the PCB. When the finished PCB was received the different components such as resistors, capacitors and microcontroller were placed on the PCB and soldered.

The microcontroller was set up to communicate with the NXT over a half-duplex RS485-line. Because this line was half-duplex, and both devices could not send data at the same time which would cause collisions, a protocol was implemented that defines the NXT as the bus-master. The microcontroller is only allowed to transmit when the NXT gives it permission.

The microcontroller has been programmet to continously make sensor readings and receive data from the Bluetooth-dongle. This is stored and transmitted to the NXT when requested.

In addition a new operation system, FreeRTOS, has been implemented on the NXT. To achieve this the NXT-device itself was opened to reveal the electronics and microcontroller inside. Some wires were soldered to the main board that would allow a connection to the microcontrollers JTAG interface. In the other end the wires were connected to a programming tool. With this tool it was then possible to program the operating system and applications into the microcontroller.

Drivers to control the different modules of the NXT, like LCD-screen and RS485-module, had to be implemented. Drivers for this was available in the operating system that had been used previously, and these were transfered to the FreeRTOS platform. With all this in place the robot-application created for the FreeRTOS, which also uses FreeRTOS, was transfered to the NXT.

Finally a new communication system for the complete system has been developed. This communication is implemented in layers inspired by the OSI-model. One of the layers, the network layer, defines the lowest data unit that is communicated. This contains the address of the receiver and sender, in addition to the data to be sent. Using the addresses the dongles can forward the data to the right place.

On top of this layer a transport layers has been developed that defines two different communication protocols. One is similar to UDP in Internet's protocol cataloge, and allows simple transfer of data. The other is inspired by TCP and uses automatic retranmission of lost data.

The data these two protocols want to send are transmitted as parts of the data-field in a network frame.

Kapittel 3

Konklusjon

Denne oppgaven hadde et viktig mål om å fjerne manglene ved NXT-roboten når det gjaldt sensorer. Det var ønsket å få fire avstandssensorer i tillegg til gyro og kompass. Dette målet er nådd ved utviklingen av kretskortet som innhenter data fra disse sensorene og overføre dette til NXT.

Oppgaven satte seg også fore å fikse kommunikasjonen mellom Bluetooth-dongelen og NXT. Utviklingen av kretskortet åpnet for muligheten til å gjøre dette ved å inkludere kommunikasjonen mellom IO-mikrokontrolleren og dongelen på en full-duplex linje, fra UART til UART. Dataen blir mellomlagret i IO-mikrokontrolleren helt til NXT ber om å få oversendt dataen på RS485-linjen som er utviklet til å være kollisjonsfri.

FreeRTOS var ønsket å få inn på NXT, og dette har blitt gjort. Overgangen til FreeRTOS gjør at det blir enklere å utvikle en felles applikasjon for alle robotene, og det må ikke tenkes på at NXT-roboten er en spesialvariant og må behandles annerledes.

Den nye formen for kommunikasjon åpner for pålitelig dataoverføring og muliggjør sending av store mengder data. Den har også en ekstra fordel da det er mulig å sende meldinger direkte mellom roboter.

Alle målene som ble satt ved oppstart av oppgaven har blitt nådd, og løsningene som er laget for å nå målene fungerer godt. Roboten har tilgang til alle sensorene den trenger, kommuniserer godt med Bluetooth-dongelen, har fått et bedre operativsystem, og hele systemets kommunikasjon har blitt oppgradert.

Når det er sagt var et av målene med oppgaven at disse videreutviklingene skulle forbedre

NXT-robotens kartlegging ved å gjøre den mer nøyaktig. Resultatene viser derimot at roboten ikke uttfører kartleggingsoppgaven tilfredsstillende. Arbeidet har ikke dreid seg rundt å oppgradere applikasjonen, men det kunne vært forventet at nøyaktigheten når det kommer til posisjonsestimat og avstandsmålinger ville blitt forbedret. Økningen av antall avstandssensorer fra to til fire har løst et problem med at roboten ofte kolliderte med vegger da den ikke så disse, men det oppleves at målingene ikke er så nøyaktige som det kunne vært ønsket. Selv om roboten nå har tilgang til gyro- og kompassdata har det ikke lykkes å ta disse i bruk på en fullgod måte, og disse er per nå ikke tatt i bruk ved estimering av posisjon.

Kapittel 4

Introduksjon

4.1 Bakgrunn

Bruksområdet til roboter er nærmeste ubegrenset, det finnes knapt de oppgaver roboter i dag ikke er i stand til å utføre, og også ofte bedre enn mennesker. I mange tilfeller er de billigere i drift enn mennesker, og har derfor overtatt mange jobber. Roboter blir også tatt i bruk når oppgaven er upassende for mennesker. Dette kan være fordi de tar mindre plass og kommer til steder mennesker ikke gjør, eller fordi oppgaven eller arbeidsområdet er av en slik karakter at et menneske ville vært utsatt for fare.

Det kan tenkes scenarier der det er nødvendig med et overblikk over et område som mennesker ikke når frem til eller der ferdsel ville utgjort en trussel mot liv og helse. En slik situasjon kan være undersøkelse av ruiner etter en naturkatastrofe eller terrorangrep, eller kartlegging av en bygning som ønskes å inntas i en stridssituasjon.

I slike tilfeller kunne roboter hjulpet til med å utvikle et kart over området for å hjelpe redningsarbeidere, soldater eller andre med å ta viktige avgjørelser.

En rudimentær løsning for dette problemet har vært under utvikling av studenter ved NTNU siden 2004. Gjennom prosjekt- og masteroppgaver har studentene hvert år bygget videre på arbeidet gjort før dem. Gjennom årene har løsningen blitt videreutviklet og nye moduler lagt til det eksisterende systemet.

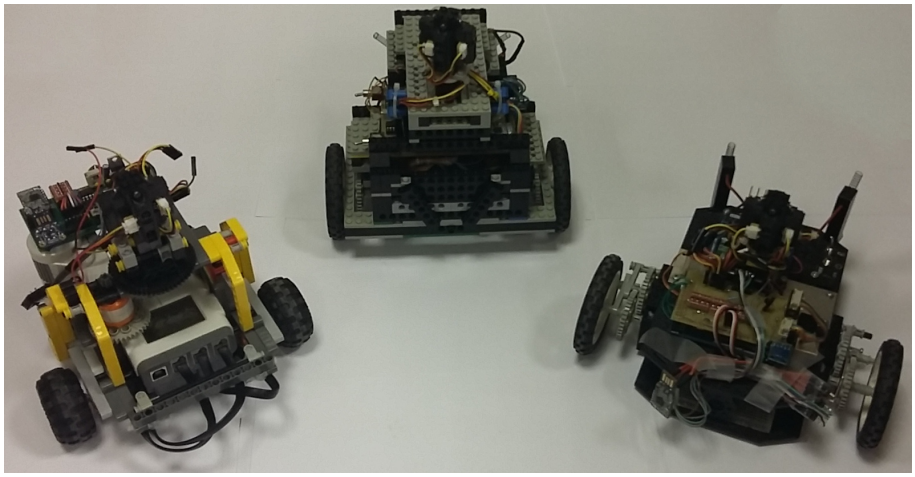
4.2 Beskrivelse av systemet

Ved oppstarten av denne oppgaven består systemet av fire roboter basert på forskjellige plattformer, og det er startet arbeid med en drone som skal kartlegge fra luften. Det er forskjellig hvor mye som er arbeidet med de forskjellige robotene og det er dermed forskjellig hvor godt de enkelte utfører kartleggingsoppgaven. Robotene bygger på noen av de samme prinsippene og metodene, men har også særegenheter som medfører sine egne problemer som må forbedres for å kunne yte optimalt.

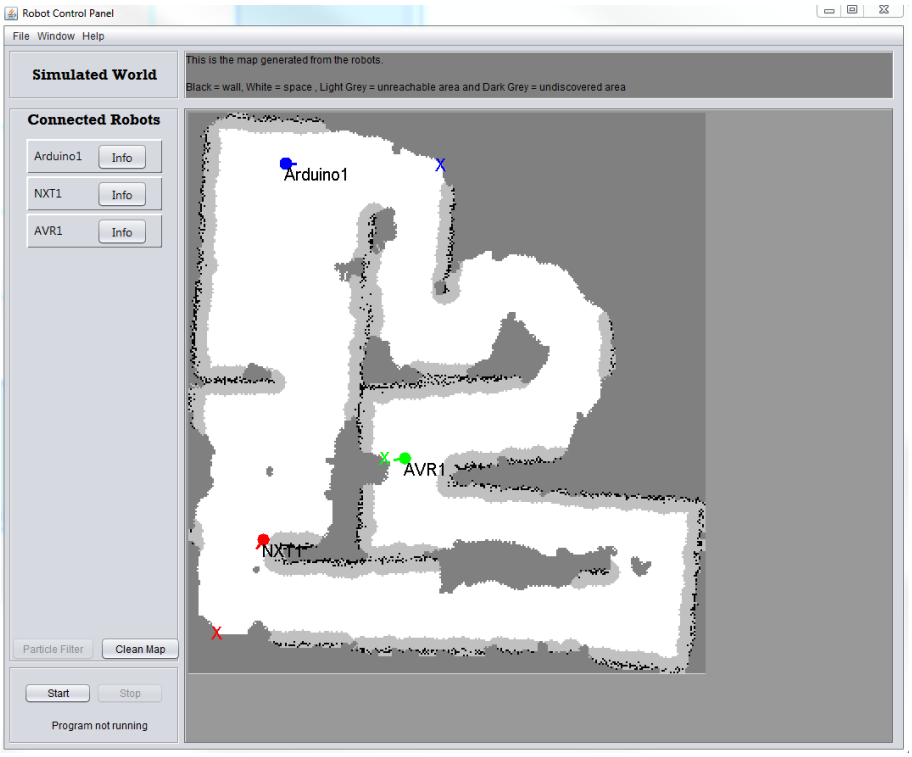
Målet er at robotene skal arbeide sammen om å utvikle et kart over området de beveger seg i. Systemet er sentrert rundt en applikasjon som kjører på en server-datamaskin. Avstandssensorer som robotene er utstyrt med fanger opp omgivelsene og disse målingene blir sendt til serveren sammen med posisjonen der de ble tatt. Basert på dette tegnes et kart i sanntid på serveren. Serverapplikasjonen inneholder en algoritme som basert på den tilgjengelige dataen avgjør hvor robotene skal sendes videre for å kartlegge de delene som enda ikke er registrert.

Kommunikasjonen mellom robotene og serveren skjer via Bluetooth Low Energy. Hver robot er utstyrt med en Bluetooth-dongel som serveren kan koble seg til for å sende kommandoer og motta målinger.

To av robotene er basert på sett fra Lego som muliggjør utvikling av egne programmerbare roboter. En annen er bygget rundt en Atmel AVR mikrokontroller, mens den siste har en Arduino som kjerne.



Figur 4.1: Roboter. Fra venstre: NXT, AVR, Arduino

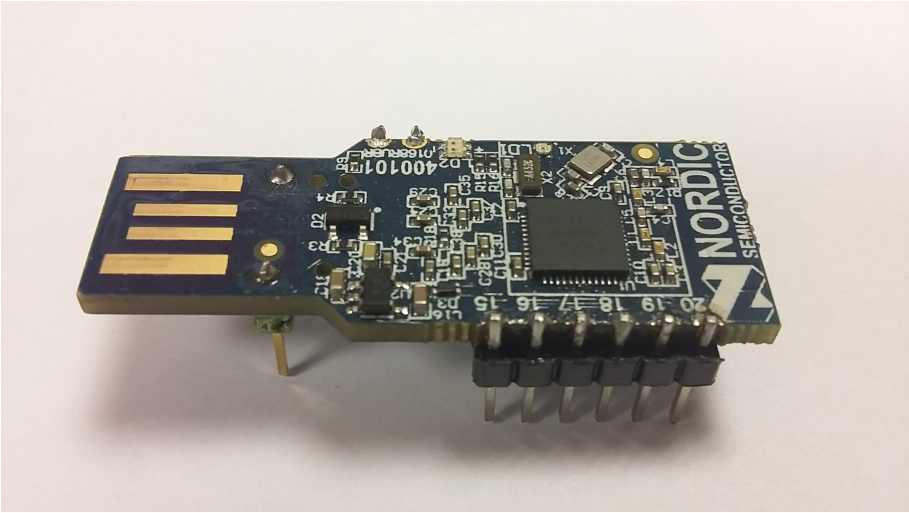


Figur 4.2: Server-applikasjon

4.2.1 NXT-robot

Den ene Legoroboten er basert på Mindstorms NXT-settet til Lego. Settet er brukt til å bygge en robot som beveger seg ved hjelp av to hjul. Midt på roboten er det bygget et tårn der to avstandssensorer er montert. Avstandssensorene er tilkoblet to av sensorportene på styringsenheten, mens en annen sensorport er koblet med kabel mot Bluetooth-dongelen. De tre servo-motorene som roterer sensortårnet og hjulene er koblet inn på de tre motorinngangene NXT har.

Bluetooth-donglen er den samme for alle robotene, og er satt opp til å motta data fra robotene på en UART. Data mottatt her sendes videre over Bluetooth til serveren. Den eneste interaksjonen som er mulig med den underliggende mikrokontrolleren i NXT er med de signalene som er eksponert mot omverdenene via sensorportene. Ingen av disse er tilkoblet UARTen inne i NXT og det har derfor vært nødvendig å bruke et RS485 signal som kan hentes fra en av sensorportene. Dette konverteres ved hjelp av elektronikk på et eksternt kretskort til logiske nivåer som leses av dongelens UART.



Figur 4.3: Bluetooth-dongel

4.3 Problemstilling

Denne oppgaven vil i hovedsak dreie seg rundt videreutvikling av NXT-roboten. Tidligere arbeid med denne har avdekket en rekke problemer som må løses. Se Lien (2016) [3] for mer informasjon enn det som presenteres her.

Det er laget et kretskort som fungerer som kommunikasjonslink mellom NXT og Bluetooth-dongel, men dette er ikke godt nok. Hovedproblemet er at begge parter kan sende data samtidig, noe som vil føre til en kollisjon på RS485-linken.

Applikasjonen som er laget for robotene inneholder en posisjoneringsalgoritme basert på kompass- og gyroskop-målinger. NXT-roboten er ikke utstyrt med dette og posisjonen bestemmes bare på basis av motorenkodere.

Operativsystemet på roboten er spesiallaget for NXT-plattformen og er i stor grad forskjellig fra operativsystemet de andre robotene bruker. Derfor har den felles applikasjonen blitt modifisert betydelig før den kunne kjøres på NXT. Hver gang den felles applikasjonen endres må den derfor tilpasses før den kan implementeres på NXT.

Roboten er bare utstyrt med to avstandssensorer, mens de andre har fire, og applikasjonen er laget med utgangspunkt i at fire målinger er tilgjengelig. Derfor har det vært nødvendig å lure systemet til å tro at roboten faktisk utfører fire avstandsmålinger. Dette resulterer ikke bare i tregere kartlegging, men også kollisjoner siden roboten ikke har oversikt over hva som befinner seg fremfor den når sensortårnet er rotert 90 grader.

Oppgaven vil i tillegg til arbeid spesifikt rettet mot NXT-roboten ta for seg kommunikasjonen mellom roboter og server. Måten dette er gjort på i dag er uten noen form for verifikasjon om at meldingene kommer frem, og resulterer i at meldingstap eller meldingskorrupsjon ikke blir oppdaget.

I punktform kan de overnevnte problemstillingene sammenfattes:
- Kommunkasjonen mellom NXT og Bluetooth-dongel må gjøres kollisjonsfri
- Fire avstandssensorer skal implementeres
- Gyroskop og kompass skal implementeres
- Operativssystem skal endres til FreeRTOS
- En kommunikasjonsprotokoll mellom roboter og server skal utvikles

Kapittel 5

Teori

5.1 Lego Mindstorms NXT

Lego Mindstorms er en serie av byggesett for å lage programmerbare roboter. Settene inneholder en styringsenhet som er hjernen til roboten, og servomotorer og sensorer som kobles til denne. NXT er det andre i rekken av Mindstorms-sett og kom ut i 2009. NXTen har tre porter for tilkobling av servomotorer, og fire for sensorer. Selve robotkonstruksjonen består ofte av selve styringsenheten og vanlige lego-komponenter.

Figur 5.1: Lego Mindstorms NXT

Roboten programmeres med et verktøy på en datamaskin og programmet overføres via en USB-kabel.

5.2 I^2C

Kommunikasjonsprotokollen I2C er laget for kommunikasjon over korte avstander mellom enheter på samme kretskort. Blant de kommuniserende enhetene er det én som er master og styrer datautvekslingen. I2C bussen består av to ledere som alle enhetene er koblet til: ett klokkesignal og ett datasignal. Slavene har hver sin 7-bits adresse som masteren sender ut på datalinjen når den vil kommunisere med dem. Etter adressen følger selve dataen som skal kommuniseres. Dersom masteren har indikert at den vil sende gjør den dét, men hvis den vil ha data fra slaven er det slaven som endrer datasignalet. Bitene på datalinjen blir avlest på hver puls på klokkelinjen.

5.3 UART

En UART er en enhet som håndterer seriell kommunikasjon. Seriell kommunikasjon er at data overføres én bit av gangen på samme linje. Mikrokontrollere inneholder vanligvis en UART. Hvis mikrokontrolleren vil sende noen byte, gis disse til UARTen som overfører dem som en sekvens av logiske nivåer på en av utgangene til mikrokontrolleren. En UART kan også avlese en slik sekvens på en av mikrokontrollerens innganger, og danne byte som mikrokontrolleren kan bruke.

5.4 RS-485

RS-485 er en spesifikasjon for hvordan data kan overføres serielt. Den fungerer godt over lange avstander og i støyete miljø. Sekvensen av bit som skal overføres sendes på to linjer som et differensielt signal: det er differansen mellom de to linjene som utgjør selve signalet. Dersom differansen er større enn –200 mV representerer dette en logisk ’1’, mens differanse større enn +200 mV er ’0’.

5.5 FreeRTOS

FreeRTOS er et operativsystem som er designet for å være lite og enkelt. Selve kildekoden omfatter bare tre kodefiler. Det er laget versjoner for 35 forskjellige mikrokontrollere.

5.6 Forbindelsesorientert kommunikasjon

Kommunikasjon mellom to enheter skjer ofte ved at de på førhånd setter opp en forbindelse før noe nyttig data overføres. I et slikt tilfelle vet begge partene at det faktisk er noen i den andre enden og at de forventer å motta data. Forbindelsen opprettholdes helt til det avtales å skru den av. Ved jevne mellomrom kan forbindelses testes for å verifisere at den andre parten fortsatt er til stede og at forbindelsen ikke er gått tapt. Ofte sender partene i en slik ordning bekreftelser på at de mottar meldingene, og dersom ingen bekreftelse blir mottatt sendes meldingen på nytt. Dette sørger for at kommunikasjonen blir pålitelig: ingen meldinger går tapt.

5.7 Bluetooth Low Energy

Bluetooth Low Energy (BLE) er en trådløs nettverksteknologi. BLE er en variant av den eldre Bluetooth-teknologien. I BLE er én enhet satt opp som en sentral, mens andre er periferier som sentralen kobler til. Periferier sender ved jevne mellomrom ut 'advertisement'-meldinger som plukkes opp av sentraler. Avhengig av bruksmåten kan periferier settes opp til å ikke være mulig å koble til, og det eneste sentralene har tilgang til av data er det som ligger i advertisment-meldingene. Den andre bruksmåten er at periferiene sender ut advertisment, og når en sentral plukker opp en slik melding kan den velge å koble til periferien. Når dette skjer vil de to enhetene utveksle data ved jevne mellomrom.

5.8 Interrupt

Interrupt er en mekanisme i prosessorer som gjør at prosessoren hopper ut av den vanlige sekvensen med instruksjoner. Et interrupt kan utløses av forskjellige hendelser som krever prosessorens oppmerksomhet. Når et interrupt skjer vil prosessoren lagre viktige verdier slik at den kan fortsette der den slapp når den nødvendige interrupt-håndteringen er utført. Interruptet gjør at prosessoren hopper til en annen del av programmet der en interrupt-rutine er lagret.

Kapittel 6

IO

6.1 Bakgrunn og valg av løsning

NXT-roboten måtte utvikles til å ha fire avstandssensorer, gyroskop, kompass og Bluetooth-dongel tilkoblet.

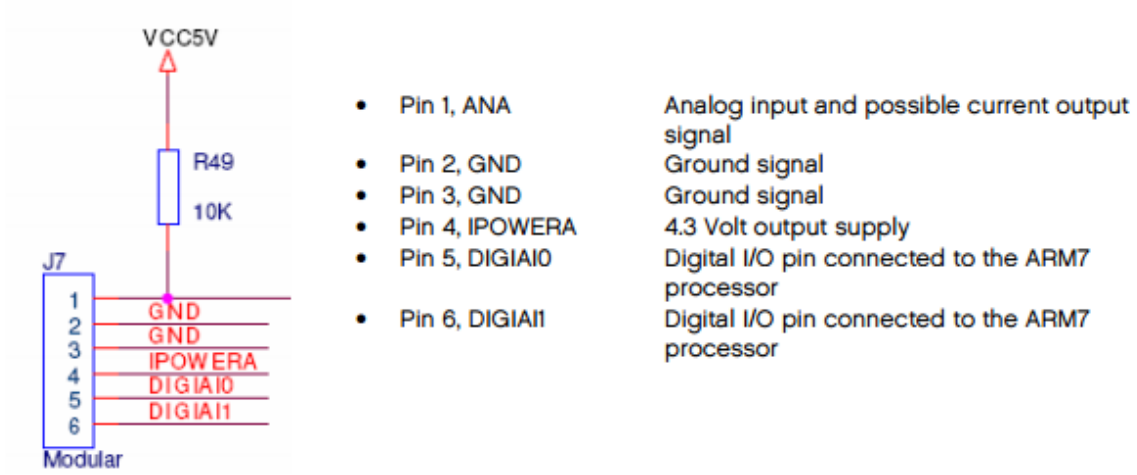
NXT er laget for å ta i bruk sensorer fra NXT-settet, og her finnes alle sensortypene som det var ønsket å implementere. Ved bruk av standard Lego-sensorer trengs i utgangspunktet én sensorport per sensor. Siden seks sensorer skal brukes og det bare finnes fire sensorporter, der én ville blitt brukt til Bluetooth-dongelen, ville det vært nødvendig med en eller annen form for multiplexing av sensorene. Riktignok er dette mulig, men hvis en slik egen løsning uansett måtte brukes, kan like gjerne egne sensorer tas i bruk. Egne sensorer er også fordelaktig siden de samme kan brukes som på de andre robotene, og man oppnå mer likhet mellom dem. Det ble derfor avgjort og ikke bruke NXT-sensorer, men å gå for en egen løsning.

Bruk av utstyr som NXT egentlig ikke er laget for å jobbe med byr på utfordringer. Ved bruk av en standard mikrokontroller ville dette vært enklere siden alt av mikrokontrollerens IO-pinner er tilgjengelig og kan kobles til eksternt utstyr som ønsket. Derimot er det mye mer begrenset hva som kan gjøres på NXT der mikrokontrolleren og all elektronikken er gjemt inne i styringsenheten. Det betyr at oppgaven måtte løses kun med signalene som eksponeres via sensorportene.

Sensportene inneholder seks signaler, se figur 6.2: 4,3V forsyning, to jord, én input til ADC, og to digitale linjer som går direkte til mikrokontrolleren i styringsenheten. De digitale linjene kan brukes til I2C, og i tillegg er sensor port nummer fire utstyrt med ekstra funksjonalitet

som muliggjør RS485 på disse signalene.

Figur 6.1: NXT med sensorporter i bunnen



Figur 6.2: Signalene tilgjengelige på sensorportene, hentet fra LEGO development kit [1]

Utfordringen var å komme opp med en løsning som bruker dette som er tilgjengelig til å koble roboten til alle sensorene og Bluetooth-dongelen. Avstandssensorene som brukes gir ut en analog spenning som varierer med avstanden som måles. Gyroskopet støtter både I2C og SPI, mens kompasset bare støtter I2C. Bluetooth-dongelen er satt opp med kode som er felles for alle robotene, og kommuniserer med logiske nivåer fra UART.

Én måte å sy alt dette sammen på som ble vurdert var å bruke en sensorport til en I2C-buss der alt kobles på. Kompass og gyroskop støtter I2C, og det kunne vært tatt i bruk en ekstern ADC med I2C grensesnitt som avleser avstandssensorene. Bluetooth-dongelen støtter også I2C og kunne blitt koblet på samme bussen.

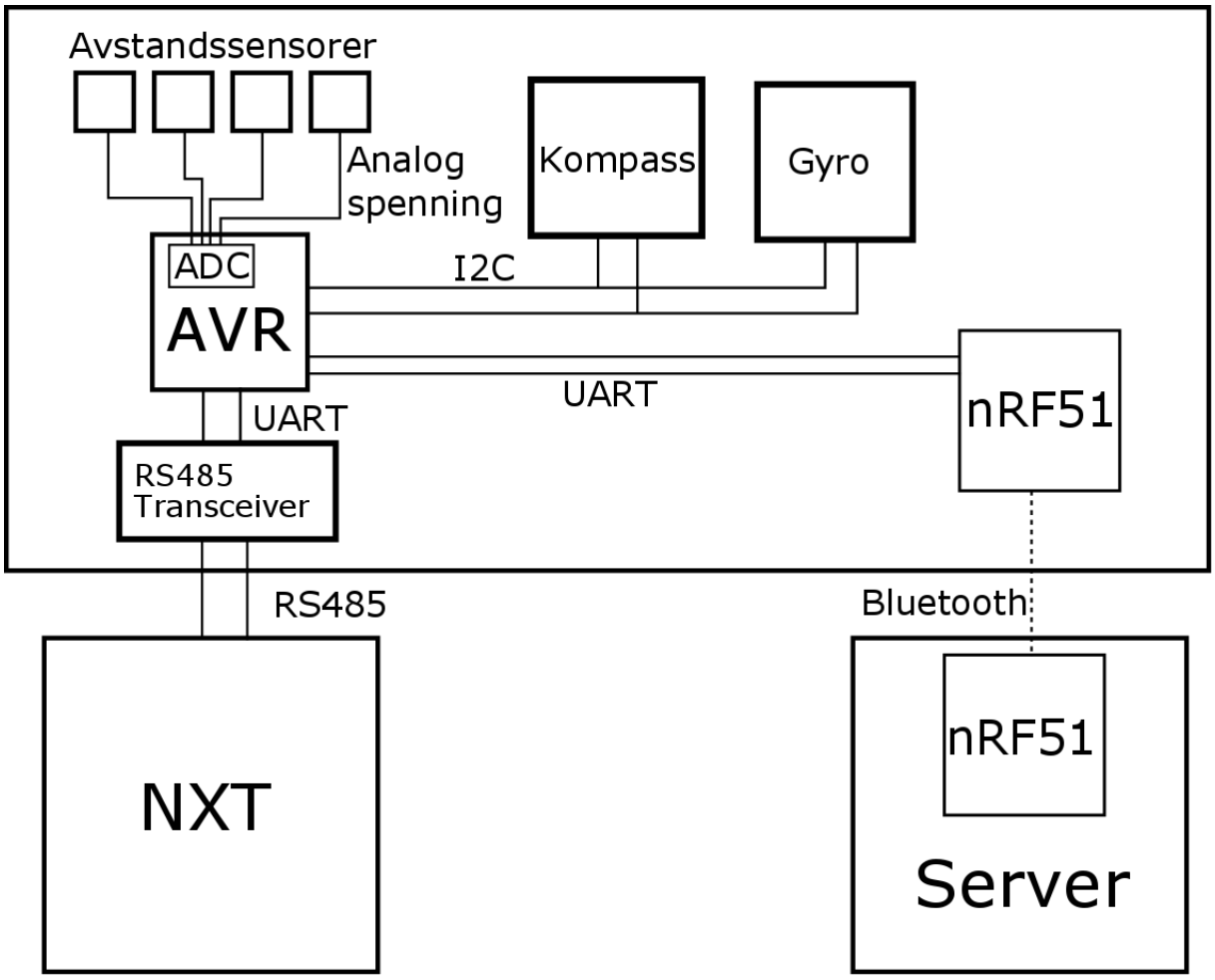
Et problem med dette er at det må lages en egen dongel-kode som endrer kommunikasjonsmåten fra UART til I2C. Å ha ulik kode på forskjellige robot-dongler var ikke ønskelig. I tillegg er I2C-grensesnittet til NXT spesielt ved at andre motstandsverdier for pullup brukes enn det som er vanlig, og tidligere forsøk med dette har ikke lykkes [3].

Løsningen som til slutt ble valgt innebærer et kretskort som fungerer som NXT sitt grensesnitt til utenomverdenen. Kretskortet inneholder en mikrokontroller som gyroskop, kompass, dongel og avstandssensorer kobles til. Kommunikasjonen mellom mikrokontrolleren og NXT skjer via RS485. En RS485-transceiver trengs derfor på kretskortet som drives av en UART i IO-mikrokontrolleren. Mellom mikrokontrolleren og Bluetooth-dongel kommuniserer UARTene hos de to med hverandre direkte.

Mikrokontrolleren kan i dette oppsettet fungere som en buffer for dataen som kommer inn over Bluetooth og deretter fra dongelens UART. Tidligere har RS485 linjen fra NXT vært brukt til å kommunisere direkte med dongelen, og da har det vært et problem at NXT og dongel prøver å sende data samtidig, som fører til kollisjon. Med den nye løsningen kan data fra dongelen mellomlagres i mikrokontrolleren og ikke videresendes før NXT ber om å motta den, og dermed ikke sender egen data samtidig som den forventer å motta noe.

Enda et problem som løses er at dongelen på én av pinnene signaliserer om den er tilkoblet en sentral, og dette signalet har det tildligere ikke vært mulig å føre frem til NXT. Nå derimot kan dette leses av IO-mikrokontrolleren og videresendes til NXT på samme måte som den andre dataen.

Figur 6.3 viser et blokkskjema for den planlagte løsningen.



Figur 6.3: Blokkskjema for kretskortet

6.2 Valg av mikrokontroller

Den planlagte løsninger satte krav til mikrokontrolleren som skulle brukes. Listen nedenfor viser periferiene mikrokontrolleren måtte ha.

2x UART	Én mot BT-dongel, én mot RS485-transceiver
1x I2C	For gyro og kompass
4x ADC-kanal	For avstandssensorer
2x GPIO pinner	Én til transceiverstyring og én som leser dongelens status

Tilgjengelighet og erfaringer gjorde at en Atmel mikrokontroller var ønsket. Ved å mate parameterene over inn i Atmels produktvelger [1] ble en liste med passende mikrokontrollerer generert. Søket tar ikke hensyn til at de ønskede periferiene i blant bruker samme pinner og

[1] `http://www.atmel.com/products/selector_overview.aspx`

ikke kan operere samtidig, så databladene til de enkelte måtte undersøkes for å passe på at dette ikke er tilfellet. I tillegg ble det sett på som et pluss med så få pinner som mulig, og through-hole montering var foretrukket grunnet enklere lodding. Listen under oppsummerer mulighetene.

Mikrokontroller	Pinner	Kommentarer
Attiny841/441	14	Ingen intern ADC. Surface-mount
Atmega164A	40	Through-hole
Atmega1284	40	Through-hole
Atmega644PA	40	Through-hole

Attiny-alternativet har fordelen av at mikrokontrolleren er liten og få pinner blir stående ubrukt. På en annen side kommer den bare i surface-mount pakker, og den har ingen intern ADC. Hvilket betyr at en ekstern ADC-chip med I2C kommunikasjon mot mikrokontrolleren ville vært nødvendig.

De andre alternativene ble til slutt foretrukket. Mellom disse er det få forskjeller, i hovedsak størrelse på program- og dataminnet. Atmega164 var lett tilgjengelig og ble valgt. Minnet på denne er minst av alternativene, kun 1 kB RAM og 16 kB Flash, men ble ansett som tilstrekkelig for mikrokontrollerens begrensede funksjon.

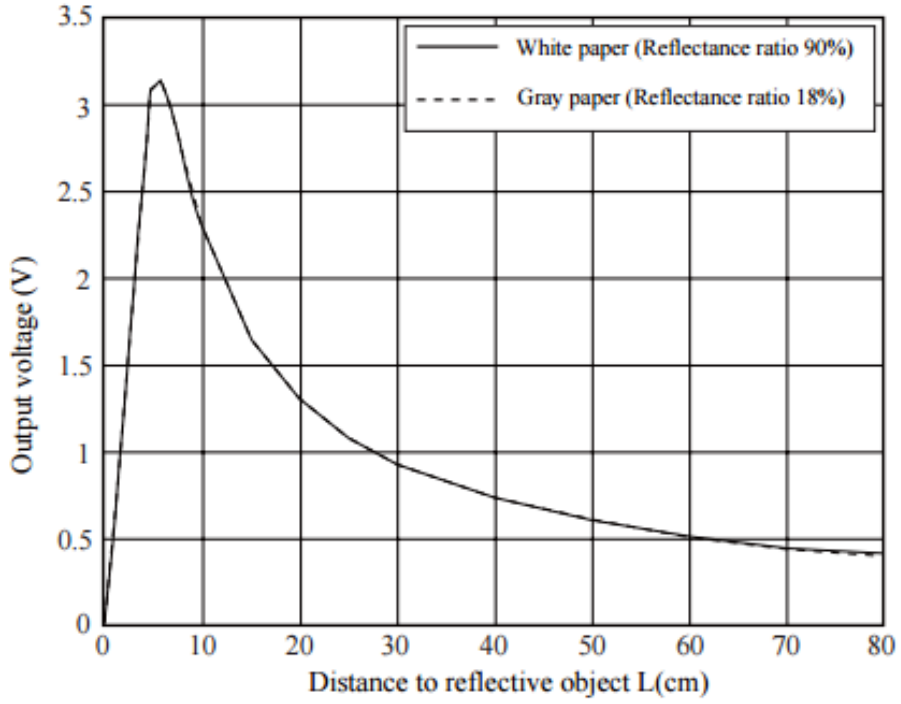
6.3 Avstandssensorer

Avstandssensorene robotene bruker er av typen GP2Y0A21YK0F fra leverandøren Sharp. Disse leverer en analog spenning som varierer med målt avstand.

Mikrokontrolleren som ble valgt har åtte innganger til ADC, så sensorene ble koblet til de fire første. Konverteringen av utgangsspenningene på sensorene til en digital verdi krever en referansespenning. De analoge spenningene konverteres til en ti-bits verdi av ADC, og referansespenningen angir hvilken spenning som er den høyeste som kan avleses, og som konverteres til ti *enere* binært, eller 1023 desimalt. Det er derfor viktig å velge en referansespenning som er høyere enn alle verdiene som forventes avlest. I tillegg er det ugunstig at referansen er mye høyere enn det som kreves, da det vil resultere i mange ubrukte digitale verdier, og man får dårligere oppløsning i det spenningsområdet som brukes.

Databladet til sensoren forteller at målingene faller i området 0 - 3,15 V. Ved nærmere inspeksjon av databladet til sensoren oppdages at maks utgangsspenning for det området den

er spesifisert for, 10-80cm, er 2,3 V ved 10cm, se figur 6.4. Mikrokontrolleren tilbyr en intern referansespenning til ADC på 2,56 V, og dette passer derfor veldig godt med spenningene som skal avleses.



Figur 6.4: Spenning mot målt avstand

Et kritisk valg som er gjort med hensyn på oppløsning er at de ti-bits digitale verdiene for enkelhets skyld lagres som åtte-bits verdier, og man mister da de to laveste bitene og betydelig oppløsning. Istedenfor at spenningsområdet deles i 1024 biter, får man bare 256. Dette gir en spenningsoppløsning på 10 mV, mens én cm endring i avstand gir minst 5 mV endring i spenning. Altså er 8-bit nok for å skille mellom omtrent annenhver centimeter. Dette gjelder ved høye avstander der spenningen varierer mindre, se figur 6.4, mens ved lavere avstander vil det være mulig å skille mellom hver centimeter. Det ble vurdert at pluss/minus to centimeter er mindre feilmargin enn det andre deler av systemet kan levere, og fordelene med å bare bruke åtte bit ble derfor avgjørende. Den største fordelen er at kalibreringen blir mye enklere. Det må lages en tabell for hver sensor som linker de forskjellige verdiene mot avstandene dette representerer. Så istedenfor å måtte bestemme hvilken avstand alle de 4 x 1024 forskjellige verdier står for, er det bare 4 x 256. Dette sparer også mye plass da størrelsen på alle tabellene sammenlagt blir 1024 byte istedenfor 4096.

Dersom systemet i fremtiden blir så nøyaktig at det er avstandsmålingene som holder det tilbake, er det da mulig å gjøre det nødvendige arbeidet for å få den fulle oppløsningen.

Siden det nå er fire sensorer var det nødvendig å bygge om tårnet de er montert på. Resultatet er en konstruksjon der fire sensorer er montert og peker i 90 grader i forhold til hverandre.

6.4 Gyroskop og kompass

Den valgte løsningen innebar at gyroskop og kompass skulle kobles til IO-mikrokontrolleren via en I2C-buss. For å få dette til måtte det tas hensyn til et par momenter. Hvilken spenning bussen skulle kjøre på var den første avgjørelsen som måtte tas. Ideelt sett burde disse sensorene forsynes med samme spenning som mikrokontrolleren, altså 4,3V, slik at de logiske nivåene som genereres på bussen har 0V for *lav* og 4,3V for *høy* og stemmer overens med mikrokontrollerens nivåer. Kompasset støtter spenninger fra 3,3-5V, men gyroskopet er ikke laget for spenninger høyere enn 3,3V, altså vil ikke en busspenning på 4,3V fungere.

Bluetooth-dongelen inneholder en spenningsregulator som leverer en 3,3 V spenning på en av utgangene når den blir forsynt med 4,3 V. En løsning som ble vurdert var å bruke denne spenningen til å drive hele kretskortet. Da ville kompasset, gyroskopet og mikrokontrolleren alle operere på samme spenning. Dette ble vurdert som en god løsning, bortsett fra at avstandssensorene trenger høyere forsyningsspenning enn dette.

Undersøkelse av mikrokontrollerens datablad førte til slutt til en overraskende enkel løsning. Inngangspenninger registreres som logisk *høy* dersom de overstiger 0,7*Vcc, eller 3,01 V i dette tilfellet. Dette betyr at med en busspenning på 3,3 V vil mikrokontrolleren likevel tolke de logiske nivåene riktig. Den genererte 3,3 V spenningen ble derfor brukt til å forsyne kompasset og gyroskopet.

Neste utfordring som måtte tenkes gjennom gjaldt pull-up motstandene på klokke- og datalinjene til I2C-bussen. I2C-enheter er bare i stand til å trekke linjene mot 0 V, ikke drive dem høye, altså må linjene kobles via motstander til busspenningen, dette trekker dem høye når de ikke drives lavt. Vanligvis brukes to eksterne 10 kOhm motstander til dette, men både kompasset og gyroskopet har disse motstandene inkludert. Kobles begge på samme buss vil motstandene komme i parallell og den totale motstandsverdien halveres. Heldigvis var det mulig å fjerne pull-up motstandene internt i gyroskopet ved å kutte to ledere på chipen det er plassert på. Da var det bare motstandene i kompasset igjen, og disse fungerer som pull-up for hele bussen.

Driverne til mikrokontrolleren for I2C ble hentet fra AVR-roboten. Grunnet forskjellig klokke-

frekvens på mikrokontrollerene måtte bussen initialiseres med annerledes enn AVR-roboten. Iniitaliseringen går ut på å sette forholdet mellom bussfrekvensen og klokkefrekvensen til mikrokontrolleren. En vanlig I2C hastighet er 100 kHz, og mikrokontrolleren har klokkefrekvensen 7,3728 MHz. Avsnitt 6.5 begrunner akkurat dette klokkevalget.

For å oppnå 100 kHz med denne klokken måtte et register settes til en verdi basert på følgende formel fra databladet:

$$\text{SCL Frequency} = \frac{\text{CPU Clock Frequency}}{16 + 2 * TWBR * Prescaler} \tag{6.1}$$

Løst med hensyn på TWBR (TWI Bit Rate Register):

$$\text{TWBR} = \frac{\text{CPU Clock Frequency}}{2 * \text{SCL Frequency} * Prescaler} - 16 \tag{6.2}$$

$$\text{TWBR} = \frac{7\,372\,800}{2 * 100\,000 * 1} - 16 \tag{6.3}$$

Resultatet blir 28,8 = 29. Altså må TWBR registeret settes til 29.

Gyroskopet skal plasseres under roboten midt mellom hjulene, så roboten er bygget litt om for å lage en anordning der gyroskopet har blitt festet. Det ble passet på at gyroskopet ble festet slik at aksene peker riktig. Kompasset skal stå på selve kretskortet og under designet av kretskortet ble det tatt hensyn til at kompasset også blir stående riktig retning.

6.5 UART

Siden mikrokontrolleren skal bruke UART, én mot dongelen og én mot RS485-transceiveren, er det kritisk at den har en presis klokke med en spesifikk frekvens. UART er asynkron kommunikasjon, der partene ikke er forbundet med et klokkesignal som angir når bitene skal avleses. Mottakeren må derfor på forhånd vite hvilken hastighet dataen sendes med. Når en dataramme mottas synkroniseres mottakeren med start-bitet. De påfølgende bitene blir deretter samplet med en frekvens som tilsvarer den innstilte datahastigheten. Dersom klokken til mottakeren ikke er riktig vil ikke datastrømmen bli avlest på de riktige tidspunktene. Et egnet klokkesignal har en frekvens som er et multiplum av datahastigheten, og som har en eksakt periode som ikke varierer litt fra gang til gang. Dette sikrer at bitene i datastrømmen blir avlest midt i perioden, og at avlesningspunktet ikke forskyves og blir større og større for hvert bit.

Standard klokkekilde på mikrokontrolleren er en intern RC-oscillator bestående av motstander og kondensatorer. Denne har en frekvens på 8 MHz og en toleranse på ± 2% dersom den kalibreres godt. Frekvensen kan ikke deles på noen vanlige UART baud-rater, og toleransen er heller ikke god nok. Derfor er en ekstern klokkekilde nødvendig. Kilden som ble valgt er en krystall-oscillator med frekvens 7,3728 MHz. Dette krystallet er spesifikt laget for å passe med vanlige baud-rater. Tabellen under viser frekvensens forhold til et par vanlig Baud-rater.

Baud rate (bit per sekund)	Faktor
38400	192
115200	64
230400	32

Hvilken baud rate som må brukes avhenger av mengden data som sendes. Følgende estimat av datamengde mellom NXT og mikrokontrolleren ble satt opp. Gyroskop- og kompassdata utgjør 6 byte hver, da X,Y og Z verdiene er på 16 bit. Mengden Bluetooth-data er et grovt anslag da mengden vil variere avhengig av meldingene som til enhver tid sendes. Denne valgte mengden er for sikkerhets skyld satt en del høyere enn det som kan forventes.

Meldingstype	Datamengde	Periode
Bluetooth data	50 byte	200 ms
Gyro-data	6 byte	30 ms
Kompass-data	6 byte	30 ms
Avstandsmålinger til NXT	4 byte	30 ms
Dongel-status	1 byte	30 ms

Totalt blir dette 816 byte, eller 6528 bit, per sekund. Dette estimatet inkluderer ikke eventuell kommunikasjons-overhead, men for enkelhets skyld sees det bort fra det i denne grovregningen. Minstekravet til Baud rate blir da også 6528.

En standard baud rate over dette er 38400. I praksis ønskes så høy Baud rate som mulig for å få en responsiv kommunikasjon der det ikke er fare for at linjen blir kvelt av mye trafikk.

På UART Mellom IO-mikrokontrolleren og Bluetooth-dongelen er det bare Bluetooth-dataen som overføres, og kravet til Baud rate er her lavere.

Dongelens UART har tidligere vært brukt med en Baud rate på 38400, og dette ble fortsatt ansett å være akseptabelt. Kommunikasjonen mot NXT ble valgt til å være på 230400 baud. Dette er veldig mye høyere enn det estimatet tilsier at er nødvendig, men det ble ikke sett noen negative sider ved å gå høyere. Baud ratene for de respektive UARTene velges ved å i

mikrokontrolleren sette en spesifikk verdi i UBBRn, Uart Baud Rate Register. Denne verdien spesifiserer hvor mange klokkepulser det går mellom hver gang datasignalet samples. Hvilken verdi som skal brukes finnes med formelen under fra mikrokontrollerens datablad.

$$UBBR = \frac{fosc}{16BAUD} - 1 \tag{6.4}$$

Med fosc = 7,3728 MHz og BAUD = 230400 blir UBBR = 1. Resultatet blir et rundt tall fordi klokken som er valgt er laget for å være et multiplum av UART-hastigheter, og alt blir da helt nøyaktig.

6.6 Realisering av løsning på kretskort

Da løsningen beskrevet i de foregående avsnittene var definert, ble denne utviklet på et prototype-kort. Et testprogram ble lagt inn på mikrokontrolleren og testet avlesningen av alle enhetene som var koblet til. Da ble det bekreftet at løsningen fungerte: alle sensorverdiene ble avlest korrekt, og Bluetooth-dongelen kunne kommuniseres med.

Den neste oppgaven var å designe og produsere et kretskort basert på løsningen. For å utvikle designet ble CAD-programvaren EAGLE benyttet. Første skritt var å tegne et skjema der alle komponentene legges inn, og deres innganger og utganger kobles sammen i henhold til den oppsatte løsningen. Resultatet kan sees i figur 6.8.

Videre overføres dette skjemaet til en annen del av programmet som brukes til å plassere komponentene fra skjemaet der de fysisk skal være på kretskortet. Når dette gjøres blir fotavtrykkene til alle komponentene i skjemaet lastet inn, og så er oppgaven og plassere disse på en god måte. Deretter var det mange iterasjoner med legging av kobbersporene som kobler komponentene sammen, og omplassering av komponenter når det ikke var mulig å rute sporene til riktig sted uten å krysse et annet. Sluttresultatet kan sees i figur 6.9.

De resulterende design-filene ble sendt til en produsent av kretskort[2] og var ferdig to uker senere. Alle komponentene ble så loddet på og kortet ble testet for å bekrefte at alt fungerte.

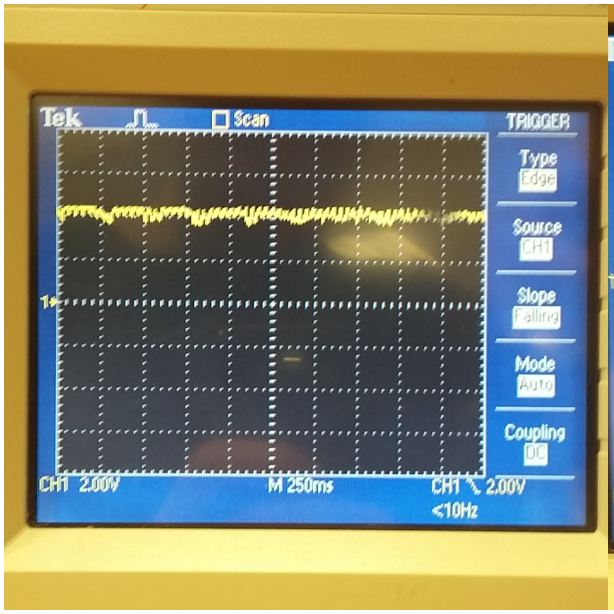
Da dette kortet hadde blitt brukt et par uker var det oppdaget en del punkter som kunne forbedres. For det første ville det være enklere å få kortet til å sitte fast på roboten hvis det

[2] pcbway.com

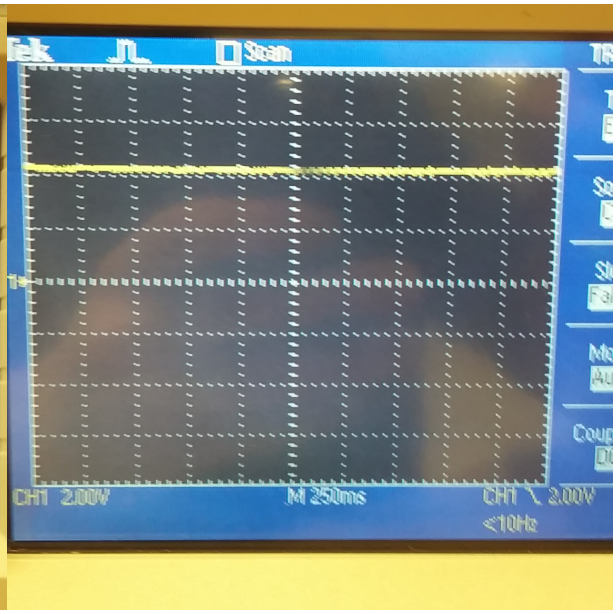
hadde andre dimensjoner. Under designet var 50x100mm brukt, men det ville være bedre om dette ble endret for å følge samme dimensjoner som legobrikker. Legobrikker har størrelse som er multiplum av 0,8 mm. Ved å basere kortet på dette er det enklere å bygge en ramme av Lego for å holde kortet på plass. I tillegg ville et par hull med dimensjoner som passer med Lego-plugger også gjøre at kortet holdes fast enda bedre. Designet ble derfor endret til å ha lengde og bredde på 48x96 mm, og to hull ble lagt til med 4,8 mm diameter og plassert riktig sted i forhold til Lego-dimensjonene.

Det ble også oppdaget at forsyningsspenningen på kortet ikke holdt seg stabil når avstandssensorene ble koblet til. Disse sensorene er avhengige av en stabil og ren forsyningsspenning for å gi nøyaktige verdier. Man så tydelig at sensormålingene var unøyaktige, figur 6.7.

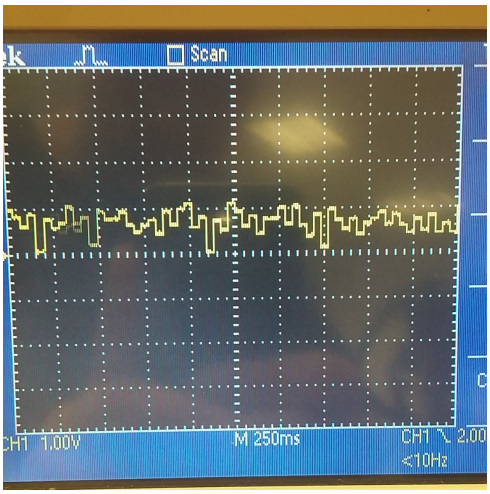
Problemet var at avstandssensorene trekker store mengder strøm hver gang de gjør en måling, hvert 40. ms, men da bare for en kort tid. Disse små plutselige økningene i strømbruk gjorde at spenningen varierte. Kondensatorer plassert nærme hver sensortilkobling kan hjelpe på dette ved å fungere som et lokalt spenningslager som tømmes når sensorene krever det, og fylles opp ellers, og jevner ut forsyningsspenningen. Derfor ble det oppdaterte kretskortdesignet endret til å ha en 470 uF kondensator ved hver sensortilkobling. Se figur 6.5 og 6.6 som viser den støyete og utjevnete forsyningsspenningen.



Figur 6.5: Forsyning før



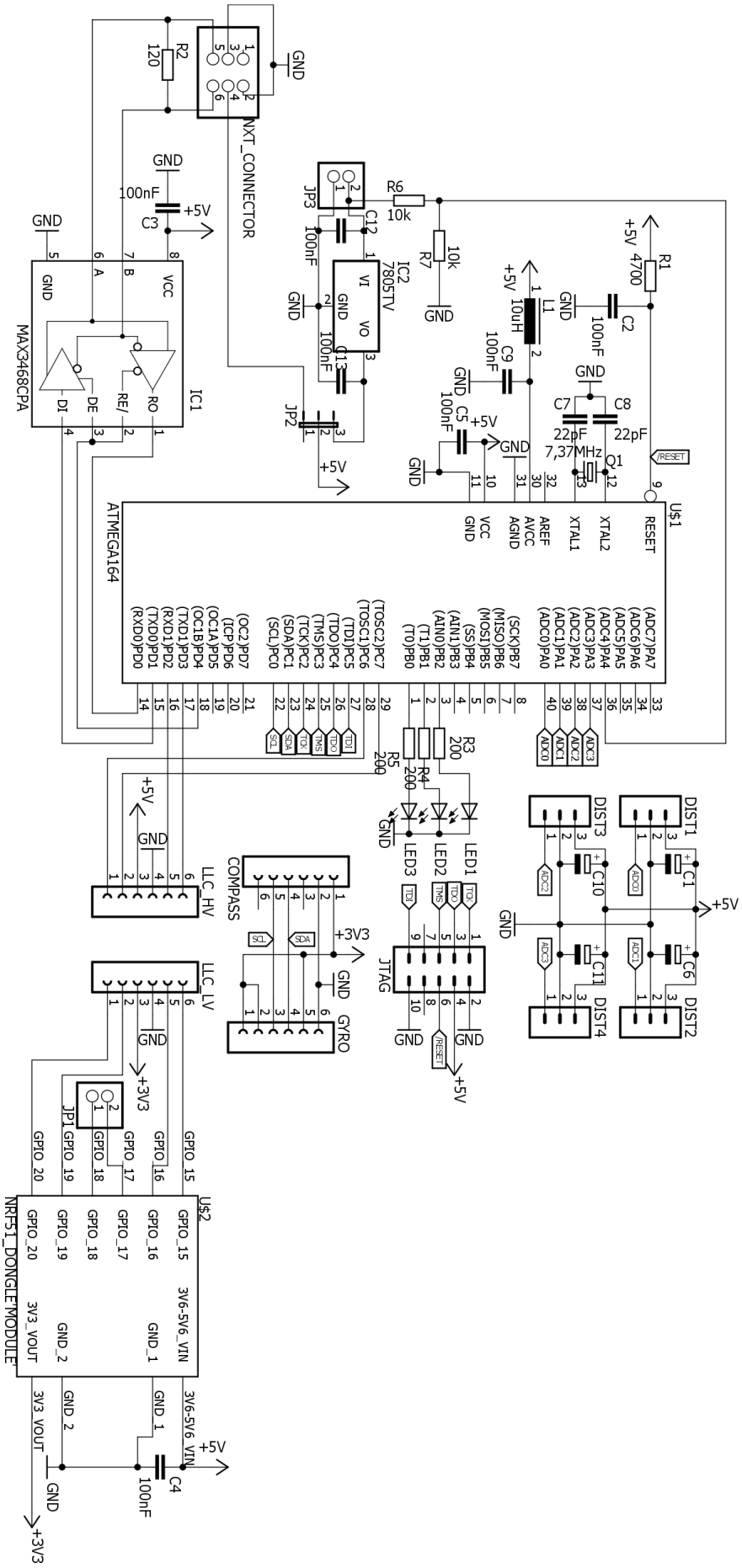
Figur 6.6: Forsyning etter



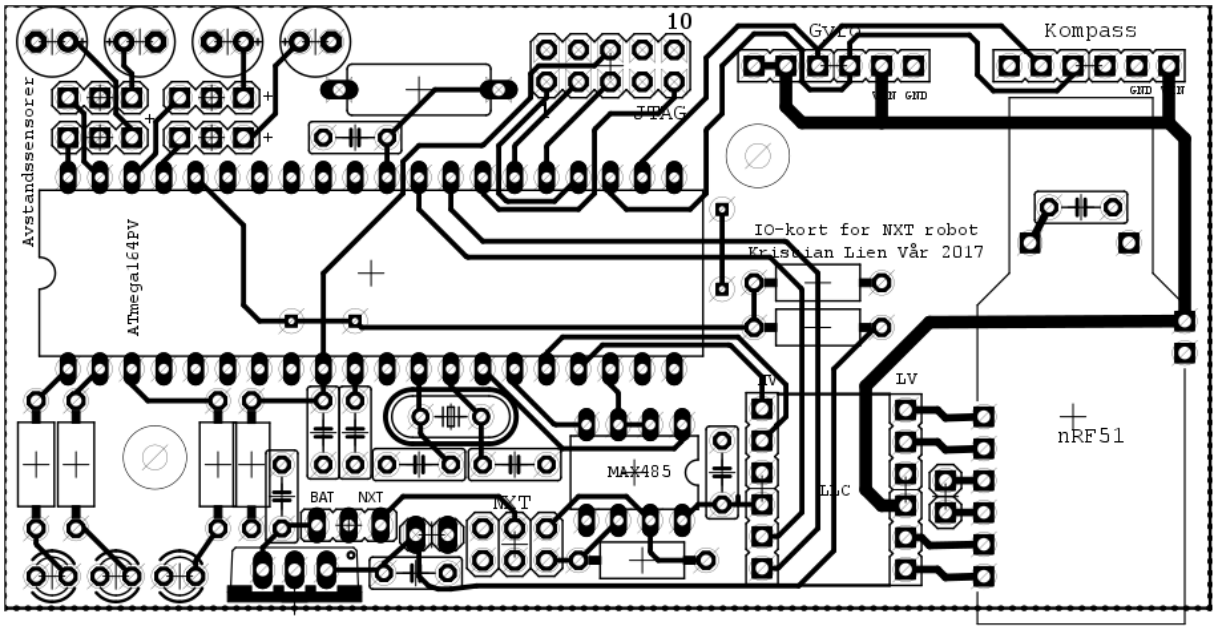
Figur 6.7: Ustabil avstandsmåling

Når strømproblemet dukket opp ble de andre komponentene også vurdert med tanke på hvor mye strøm de bruker. Tre lysdioder er plassert på kretskortet for å kunne signalisere forskjellige tilstander, men disse trekker en viss strøm. Hvordan disse spiller inn på forsyningen og avstandsmålingene ble derfor testet. Det ble observert en forverring av forsyningskvaliteten når alle LEDene var skrudd på, men denne effekten var minimal i forhold til problemene med avstandssensorene uten kondensatorer. Det ble derfor vurdert at ingen endring var nødvendig når det kom til LEDene. Men det er absolutt noe å ha i bakhodet ved utviklingen av applikasjonen: moderat diodebruk kan være en fordel.

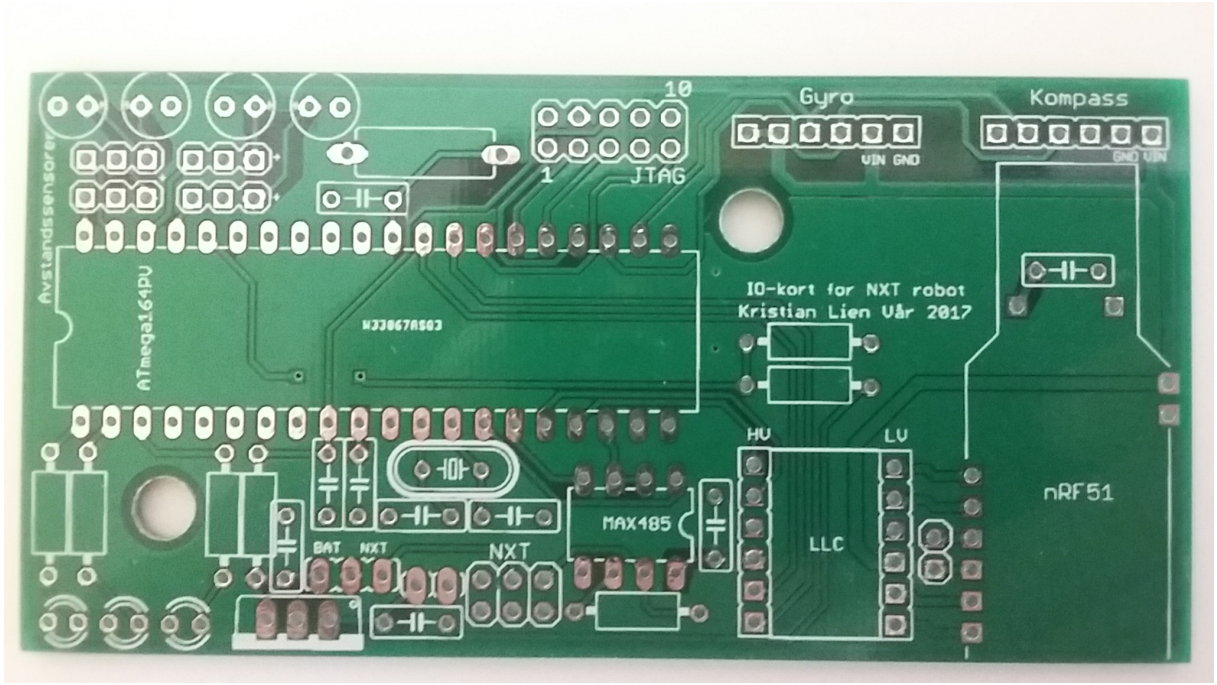
Et nytt kretskort ble produsert basert på det oppdaterte designet, og det har ikke vært noen problemer med dette.



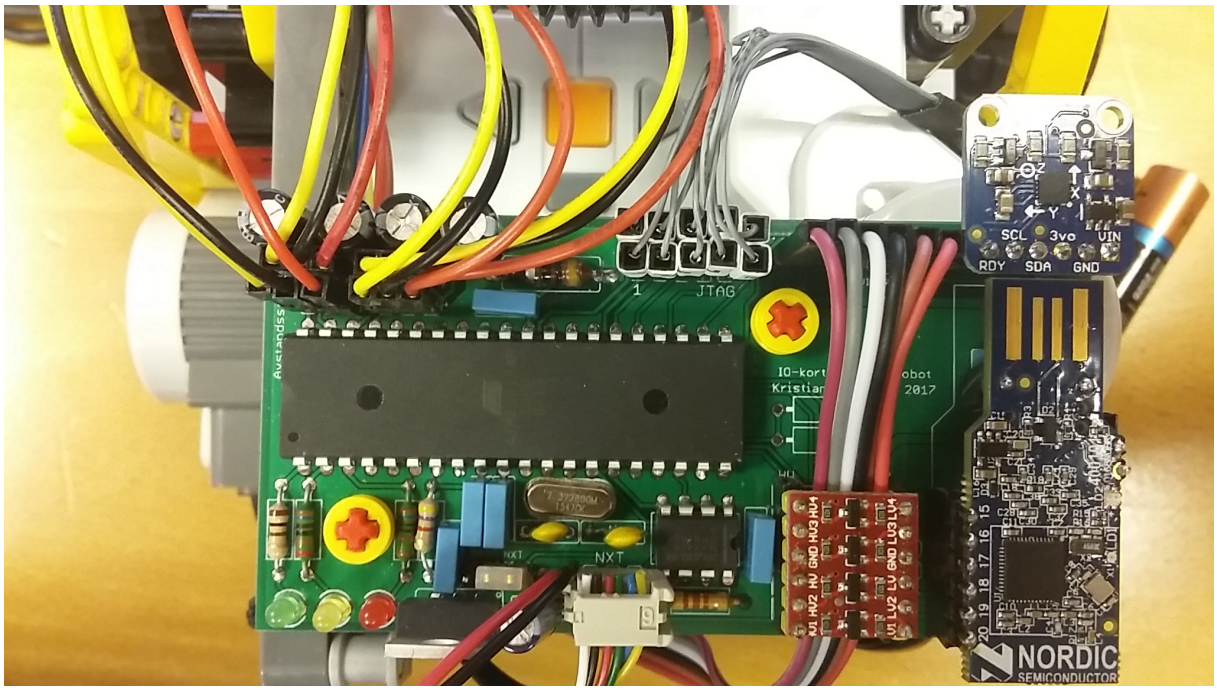
Figur 6.8: Kretskort-skjema



Figur 6.9: Kretskort-layout



Figur 6.10: Kretskort



Figur 6.11: Kretskort med alle komponenter loddet på

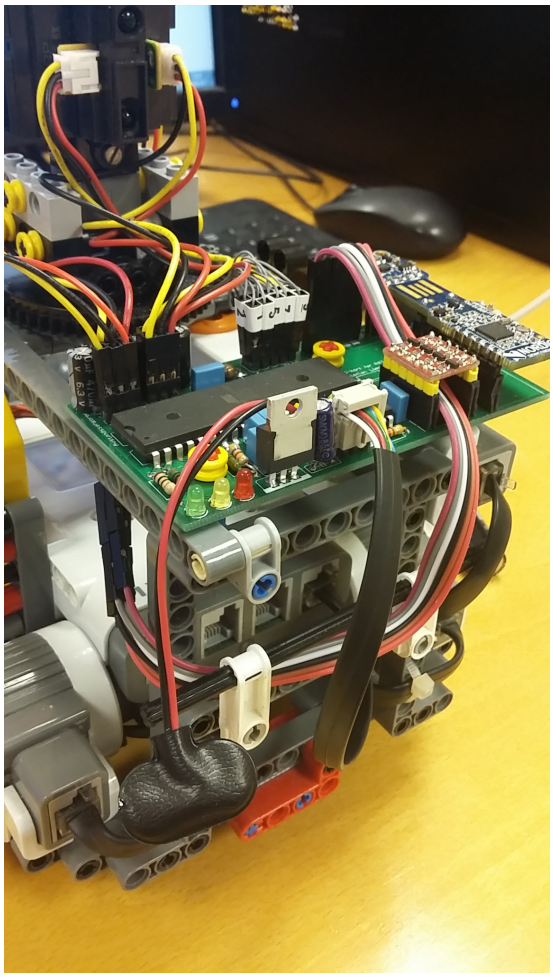
6.7 IO-NXT kommunikasjon

Mikrokontrolleren på IO-kretskortet og NXT kommuniserer med den oppsatte løsningen på en RS485-linje for å overføre sensorverdiene og Bluetooth-meldinger. Det har vært nødvendig å utvikle en protokoll for hvordan denne kommunikasjonen foregår slik at det ikke oppstår kollisjoner, og slik at begge parter er enige om hva slags informasjon det er som overføres.

6.7.1 Linjedeling

RS485-kommunikasjon foregår på to linjer og det er spenningsdifferansen mellom disse to linjene som utgjør signalet. Siden det bare er ett slikt par som forbinder NXT og IO er kommunikasjonen halv-duplex. Det vil si at kommunikasjon kan skje i begge retninger, man ikke samtidig. Dersom begge parter prøver å sende noe samtidig vil det oppstå kollisjon og tapte meldinger. Altså må begge partene dele på kommunikasjonslinjen, og det må logikk på plass for å styre hvordan denne linjedelingen skal skje.

For å løse dette er NXT satt opp som master på kommunikasjonslinjen. Med andre ord er det



Figur 6.12: NXT med tilkoblet IO-kretskort

NXT som styrer hvem som har lov til å sende data til enhver tid. Når systemet starter er det NXT som eier linjen og har lov til å sende data som den vil uten å bekymre seg for kollisjoner med data kommer den andre veien. IO-mikrokontrolleren får bare lov til å sende når den mottar en kommando fra NXT som ber den om det.

Det er fire forskjellige grunner til at NXT sender en melding til IO-mikrokontrolleren:

1. Når den vil motta Bluetooth data
2. Når den vil motta sensorverdier
3. Når den vil skru på/av LEDs på kretskortet
4. Når den vil sende Bluetooth data

De eneste gangene IO-mikrokontrolleren har lov til å sende data, er etter at den har mottatt en av de to første meldingene i listen over. Når NXT har sendt en av disse er den klar over

at innkommende data vil komme, og sender derfor ikke noe mer før svaret på meldingen er mottatt.

På denne måten er det sikret at bare én sender data om gangen og kollisjoner unngås.

6.7.2 Meldingsenkoding og feilsjekk

Når meldinger sendes over et serielt medium må det, dersom meldingene ikke har faste lengder, være en måte å bestemme hvor én melding slutter og en annen begynner. I utgangspunktet leser mottakeren bare én og én byte etterhvert som de kommer, og så er det opp til denne og tolke byte-strømmen.

Én mulighet å enkelt skille meldinger på er ved å sende all data ASCII-enkodet. ASCII definerer 128 verdier, der 95 av disse er alle de store og små bokstavene i alfabetet, tallene 0-9, og andre vanlige tegn. Resten er spesialtegn og inkluderer tegn som kan brukes til å kontrollere flyten i en kommunikasjon. Med dette kan altså vanlig tekst sendes, og spesialtegnene kan brukes for å signalisere at meldingen er slutt og mottakeren kan da tolke bokstavene den har mottatt.

Hva om dataen som sendes ikke er tekst, men arbitrær binærdata, slik som avleste sensorverdier? Det er mulig å enkode binærdataen som tekst, for eksempel hvis en sensor måler 123 kan ASCII-tegnene for 1, 2 og 3 sendes, eller siden 123 er 0x7B heksadesimalt er en mulighet å sende tegnene for 7 og B. Ulempen med dette er at det ikke er veldig effektivt. Istedenfor å sende tallet 123 som får plass i én byte, sendes én byte for hvert av tegnene, altså dobbelt og tredobbelt så mye data i dette tilfellet.

Kommunikasjonslinjen vil overføre data ofte siden sensorverdiene ønskes oppdatert så hyppig som mulig. På grunn av den dårlige effektiviteten ble det avgjort og ikke bruke ASCII-enkoding av dataen.

Utfordringen var altså at det trengtes et skilletegn for å signalisere slutt på melding og at mottakeren skal tolke dataen som er kommet inn. Med andre ord må én byteverdi settes av til dette formålet. Denne verdiene har ikke lov til å forekomme midt i en melding da dette vil få mottakeren til å tolke ufullstendig data. En sensor kan gjøre en måling som inneholder en vilkårlig byteverdi, så uansett hvilket skilletegn som velges kan det ikke unngås at dette også forekommer i data fra sensorene.

Løsningen på dette ble å bruke en enkoding kalt Consistent Overhead Byte Stuffing (COBS). [4]. Utviklerne av denne enkodingen sier:

> Byte stuffing is a process that encodes a sequence of data bytes that may contain 'illegal' or 'reserved' values, using a potentially longer sequence that contains no occurrences of these values. The extra length is referred to here as the overhead of the encoding.

Ved bruk av byte stuffing kan altså forekomster av det reserverte skilletegnet i en melding erstattes med noe annet. COBS gjør dette ved å bruke '0' som skilletegn, og enkoder meldingen slik at alle nuller forsvinner. Metoden som brukes er veldig effektiv: antallet overhead bytes er bare én per 255 byte data, uansett hvor mange nuller som befinner seg i dataen.

COBS får dette til ved å legge til en byte på starten av dataen. Denne byten inneholder antall etterfølgende byte som *ikke* er lik null, pluss én. Det kan sees på som at den innledende byten peker på den første nullen i opprinnelige dataen. Deretter erstattes hver null i dataen også av antall etterfølgende byte som ikke er null, pluss én. Tabell 6.1 viser et enkelt eksempel.

	1	3	7	0	9	11	13	
4	1	3	7	4	9	11	13	
4	1	3	7	4	9	11	13	0

Tabell 6.1: COBS eksempel

Den øverste raden inneholder den opprinnelig dataen som ønskes sendt. I midten er dataen etter at den har blitt COBS enkodet og nullen tatt bort. Den nederste raden inneholder dataen som faktisk blir sendt, med en null lagt til på slutten for å signalisere slutt på melding.

Når mottakeren mottar dette og avleser nullen, kan dataen enkelt dekodes igjen ved å følge reglene i COBS.

Feilsjekk

For å forsikre at meldingene som går mellom NXT og IO-mikrokontrolleren er feilfrie, legges det til en verdi for feilsjekk på slutten av hver melding. Denne verdiene beregnes med en CRC-algoritme basert på innholdet i meldingen. Den spesifikke algoritmen som benyttes er utviklet av Maxim [2] og produserer en feilsjekkverdi på 8-bit. I tabell 6.2 er eksempelet fra forrige avsnitt utvidet til og også inkludere feilsjekken (rad 2), som for den gitte dataen beregnes til 57.

	1	3	7	0	9	11	13		
	1	3	7	0	9	11	13	57	
4	1	3	7	5	9	11	13	57	
4	1	3	7	5	9	11	13	57	0

Tabell 6.2: CRC eksempel

Når mottakeren beregner feilsjekkverdien fra dataen den har mottatt, og oppdager at den ikke stemmer overens med verdien i meldingen, blir dataen kastet. Det blir ikke gjort noe forsøk på å sende dataen på nytt, det er så høy frekvens på meldingene at neste vil komme nesten umiddelbart.

6.7.3 Meldingsformat

Dette avsnittet gir en oversikt over meldingene som utveksles mellom NXT og mikrokontrolleren. Hver meldingstype en part sender har blitt tildelt et identifikasjonsnummer som er det første meldingen inneholder, slik at mottakeren vet hvordan dataen skal tolkes.

Sett LED - type 1.
Sendes til IO for å kontrolleren lysdiodene. Etter den første byten som angir meldingstypen inneholder meldingen bare én byte som styrer hvilke lysdioder som skal være av/på. Formatet på bitene i denne byten er 0b1tcccryg. De tre laveste bitene r,y og g står for red, yellow og green, og dersom bitet er høyt angir det at den tilsvarende dioden skal skrus på. De tre neste bitene 'ccc' er en bit-maske for 'ryg'. Dersom en 'c' er satt høy, sier det at dioden tre bit-plasser mot venstre i byten skal endres. Det vil si at dersom 'ccc' er lik 010 og 'ryg' er lik 010, blir den gule dioden skrudd på, men de andre skrus ikke av. Når t-bitet er satt angir det toggling av diodene som er valgt med 'ryg'. I dette tilfellet har ikke 'ccc' noe å si.

Send Bluetooth - type 2.
Sendes fra NXT og inneholder data som skal sendes over Bluetooth. Påfølgende byte etter type-identifikasjonen er dataen som ønskes sendt.

Returner Bluetooth - type 3.
Sendes fra NXT og består bare av én byte: typen. Dette gir ordre til IO-mikrokontrolleren om å returnere eventuell data den har mottat fra Bluetooth-dongelen.

Returner sensorer - type 4.
Sendes fra NXT og inneholder også bare typen og får IO-mikrokontrolleren til å returnere

verdiene på alle sensorene.

Sensor data - type 1.
Sendes fra IO som svar på ”returner sensor”-melding fra NXT. Bytene som følger typen er vist i tabell 6.3. Først kommer de fire avstandssensorene, deretter gyroskop- og kompassmålingene, og den siste byten angir om Bluetooth-dongelen er tilkoblet sentral-dongelen. Rad to viser hvor mange byte de respektive feltene består av, totalt består meldingen av 17 byte pluss én for typen.

Dis1	Dist2	Dist3	Dist4	GyrX	GyrY	GyrZ	ComX	ComY	ComZ	Dongle
1	1	1	1	2	2	2	2	2	2	1

Tabell 6.3: Sensormelding format

Bluetooth data - type 2.
Sendes fra IO som svar på ”returner Bluetooth”-melding fra NXT. Etter typen-byten kommer eventuelle byte mikrokontrolleren har mottatt fra donglen og mellomlagret. Dersom ingen data er mottatt returneres bare type-byten.

6.8 Resultater

Utviklingen av kretskortet med mikrokontrolleren som innhenter all IO-data gjør at NXT nå har tilgang til alle de ønskede sensor-dataene. Roboten har gått fra å ha to til fire avstandssensorer, og i tilegg er gyroskop og kompass implementert på roboten. Per nå hentes sensorverdiene hvert 30 ms fra IO-kortet. Posisjonsestimatoren kjører hvert 30. ms, så hentingen av sensorverdiene er satt til det samme som dette. Hyppigere oppdateringer ville vært bortkastet da ingen deler av applikasjoner trenger nye verdier oftere enn dette. NXT ber om å få oversent eventuell Bluetooth data hvert 50. ms.

Med disse oppdateringsfrekvensene ble det testet å samtidig sende LED-kommandoer hvert 100 ms for å toggle en LED, og det fungerte utmerket. Det var ikke mulig å se noen ujevn frekvens på LEDens blinking, som ville tydet på propper i kommunikasjonen. Man kunne tenke seg at slike propper kunne oppstått siden flere tasker prøver å sende data samtidig, og dette må bli flettet sammen på kommunikasjonslinjen samtidig som det etter sendte forespørsler om Bluetooth- eller sensordata må ventes på svar fra IO før noe mer kan sendes.

Påliteligheten til kommunikasjonen ble også testet. Det ble sendt 5000 forespørsel om å mot-

ta sensorverdier og samtidig telt hvor mange svar som ble mottatt. Resultatet var at alle 5000 forespørsler ble sendt til IO-mikrokontrolleren riktig, og NXT fikk også svar på alle. Altså fungerer systemet som er satt opp for å unngå kollisjoner godt.

Mottatte sensorverdier lagres i NXT sin IO-driver, og det er satt opp et interface mot applikasjonen som gjøre at verdiene hentes ut med lignende funksjoner som om sensorene var direkte koblet til hovedmikrokontrolleren som på de andre robotene.

Kapittel 7

FreeRTOS

I fjor høst ble operativsystemet nxtOSEK lagt inn på NXT-roboten av Lien [3]. Dette operativsystemet er spesiallaget for NXT-plattformen og var av den grunn enkelt å overføre og implementere, og det var også derfor det ble brukt. For å få kartleggingsapplikasjonen til å kjøre i dette operativsystemet var det nødvendig å ta i bruk en rekke innfløkte løsninger som gjorde koden komplisert og uoversiktlig. I tillegg ville det være tungvindt å vedlikeholde samme applikasjon for to forskjellige operativsystemer.

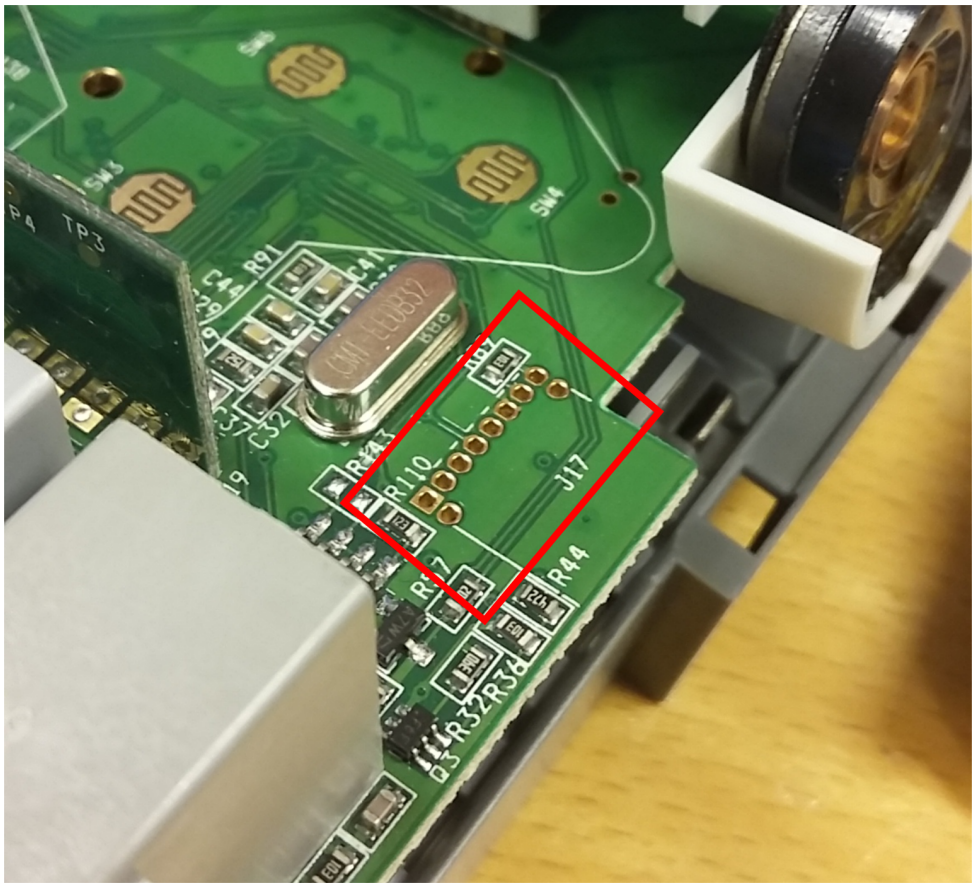
Av disse grunnene ble det avgjort å implementere samme operativsystem som de andre robotene bruker, FreeRTOS, også på NXT-roboten.

7.1 JTAG

Siden operativsystemet som hadde vært i bruk tidligere var laget for NXT-plattformen, fantes det også PC-verktøy for å overføre programmer via USB-kabel. Når overgangen gjøres til et annet operativsystem ville ikke dette lenger være mulig. For å kunne overføre selve operativsystemet og kode var det derfor nødvendig å sette opp en tilkobling til prosessoren via et JTAG-interface.

Hovedkortet til NXT-roboten er utstyrt med tilkoblingspunkter for JTAG. Disse punktene er koblet til de nødvendige inngangene på prosessoren. Se figur 7.1.

Figur 7.2 viser hvordan disse punktene skal kobles sammen med programmeringsverktøyet. Bildet er hentet fra Legos NXTs hardware development kit [1]. Til høyre er J17, som er navnet

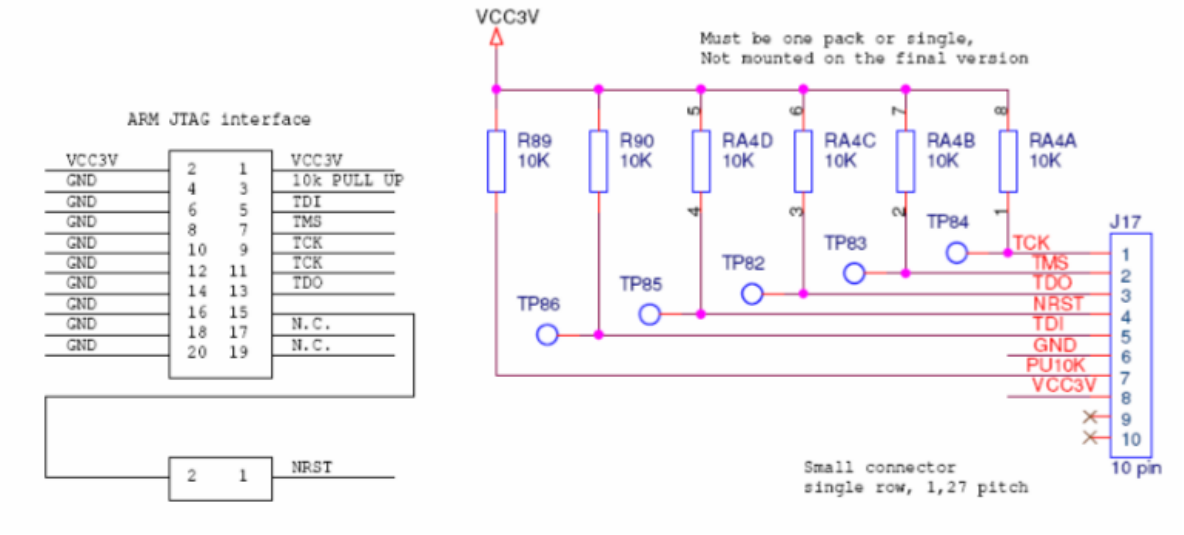


Figur 7.1: JTAG tilkoblingspunkter

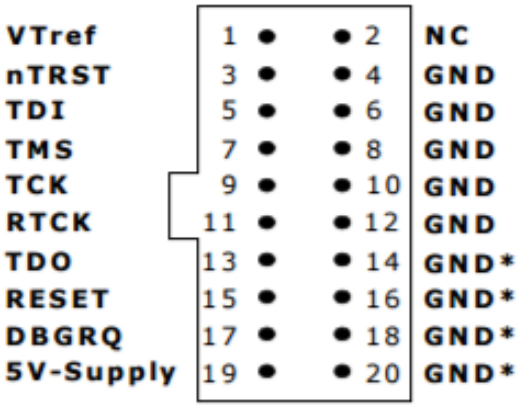
til headeren på hovedkortet, og hvert punkt har navnet på JTAG-signalet som skal kobles til der.

For å ta i bruk JTAG interfacet trengs et programmeringsverktøy for ARM-baserte systemer, og til dette er SEGGER J-LINK benyttet. Se figur 7.4. J-LINK kontakten har 20 pinner, figur 7.3, men bare 9 av disse brukes. Disse 9 tilsvarer signaler fra J17.

I henhold til disse skjemaene ble ledninger loddet på kretskortet og de andre endene koblet inn i J-LINK kontakten. Figur 7.5 og 7.6 viser resultatet.



Figur 7.2: NXT JTAG interface



Figur 7.3: J-LINK inngangsport

Figur 7.4: Segger J-LINK

Figur 7.5: Ledninger loddet til punktene

Figur 7.6: NXT med programmeringsverktøy

7.2 Tilpassing og implementering

FreeRTOS finnes i versjoner for mange forskjellige maskinvarer. Den enkleste måten å komme i gang med FreeRTOS på, og som anbefales av utviklerne, er å ta utgangspunkt i et av de mange eksempelprosjektene som finnes. Dersom det ikke finnes noen versjoner og eksempler for akkurat den maskinvaren man har, anbefales det å modifisere en versjon for en lignende maskinvare.

NXT er basert på en Atmel mikrokontroller med en ARM7 prosessor. Helt spesifikt er det

AT91SAM7S256 mikrokontrolleren som er inne i NXT. Det finnes en FreeRTOS versjon for AT91SAM7S64, altså akkurat den samme bare med 64 kB flash minne istedenfor 256 kB.

Av den grunn var det ikke mye som måtte gjøres for å få denne versjonen til å kjøre på NXT-hardwaren. Faktisk ville den fungert fint uten modifikasjoner, men for å ha tilgang til hele minnet måtte linker-konfigurasjonen endres, og en konstant i FreeRTOS-kilden måtte økes til å samsvare med den utvidede minnekapasiteten.

Eksempelprosjektet som er tatt i bruk er laget for utviklingsmiljøet IAR Embedded Workbench. Dette er et program som i utgangspunktet koster penger, men det finnes to ulike gratisversjoner med visse begrensninger. Den ene av disse har ingen begrensninger i funksjonalitet, man man er begrenset til 30 dagers bruk. Den andre har ingen tidsbegrensning, men programstørrelsen er begrenset til 32 kB. Det er den siste som er brukt under arbeidet med denne oppgaven, men begrensningen denne medfører har vært et problem, da størrelsen på det ferdige programmet akkurat ligger på denne grensen.

7.3 Drivere

Mikrokontrolleren inne i NXT trenger drivere for å ta i bruk den hardwaren NXT er utstyrt med. De modulene som var absolutt nødvendige for denne oppgaven og som det måtte implementeres drivere for var UART og I2C. UART brukes til å drive RS485 signalet som går til IO-mikrokontrolleren, og I2C brukes til å snakke med AVR-hjelpemikrokontrolleren som også finnes inne i NXT, og som styrer motorene. I tillegg ville det være en fordel og få opp SPI drivere, siden SPI brukes til å styre LCD-skjermen på roboten som er hendig å skrive debugmeldinger til. På toppen av disse måtte det også drivere til som tar i bruk de overnevnte for å utføre en funksjon. For eksempel skjermdriver som bruker SPI for å skru på og skrive til LCD-skjermen, og motordrivere som bruker I2C til å fortelle hjelpemikrokontrolleren hvordan motorene skal styres.

Drivere for alt dette var beleilig nok tilgjengelig i kildekoden for nxtOSEK, og kunne med visse modifikasjoner også anvendes i FreeRTOS.

Modifikasjonene som måtte gjøres dreide seg i hovedsak rundt interrupts. Én av de sentrale interruptkildene i mikrokontrolleren er en såkalt Periodic Interval Timer (PIT). Dette er en veldig presis timer som kan stilles inn til å aktiveres med en gitt periode. Driverne fra nxtOSEK tar i bruk denne for å gjøre periodiske oppgaver, som å kommunisere med hjelpemikro-

kontrolleren hvert millisekund. Problemet var at FreeRTOS-implentasjonen bruker dette interruptet til å drive hele operativsystemet. Hver gang interruptet aktiveres kjøres operativsystemets scheduling-algoritme som bestemmer hvilken task som skal kjøre.

Driverne måtte derfor endres til å ta i bruk et annet interrupt. Mikrokontrolleren inneholder også andre timer-interrupts, så én av disse ble satt opp til å gjøre funksjonen PIT gjorde tidligere.

En større utfordring var å få interruptene fra driverne til å passe inn i interrupt-modellen til FreeRTOS, og fungere side om side med operativsystemet. Når et interrupt aktiveres er det viktig at prosessorens og operativsystemets kontekst blir lagret, slik at programmet kan fortsette der den slapp når interrupt-rutinen har fullført. Denne kontekst-lagringen gjøres i assembly-kode, og er forskjellig i driverne og FreeRTOS. Det har derfor blitt brukt mye tid på å fikle i assembly for å få det hele til å fungere.

7.4 Applikasjon

Det neste steget etter å ha fått på plass FreeRTOS og de nødvendige driverne, var å få selve robot-applikasjonen til å kjøre. Et av målene da FreeRTOS ble tatt i bruk var at applikasjonen fra AVR-roboten, som bruker det samme operativsystemet, kunne overføres og kjøres mer eller mindre uendret.

Kildekoden fra AVR-roboten ble overført, og de få hardware-spesifikke delene av koden ble endret. Dette inkluderte IO-funksjonene, som hos NXT ligger i IO-driveren som henter data fra det eksterne kretskortet.

Dette var alt som måtte til for å overføre applikasjonen. Roboten ble skrudd på og startet med kartleggingen: alt virket som det skulle.

Kapittel 8

Kommunikasjon

Kommunikasjonen mellom robotene og serverapplikasjonen er en viktig del av systemet, uten dette får ikke systemet utført sin oppgave. Dataoverføringen skjer trådløst over Bluetooth Low Energy. Alle robotene er utstyrt med en Bluetooth-dongel satt opp som en BLE periferi, og server-datamaskinen har en dongel implementert som en sentral.

Denne oppgaven tok seg fore å endre hvordan robotene og serveren tar i bruk donglene og kommuniserer med hverandre. Hovedmålet med dette var å implementere en pålitelig protokoll som sikrer at viktig data faktisk blir overført, og at større meldinger kan sendes.

8.1 Kommunikasjonsform ved oppstart

For å ha et sammenligningsgrunnlag under den påfølgende beskrivelsen av den nye kommunikasjonsmodulen, presenteres her hvordan datautvekslingen foregikk med det originale oppsettet.

Mellom serverapplikasjonen og sentral-donglen i USB-porten var det implementert et kommandosett som bestemte hva donglen skulle gjøre. For at donglen skulle vite om den innkommende dataen var en kommando eller om den skulle sendes til en robot, måtte det alltid sendes et spesialtegn før hver kommando.

Under er vist en typisk oppstartssekvens. Her bes donglen skanne etter robot-dongler og så litt senere om å stoppe skanningen. List-kommandoen gjør at donglen sender de oppdagede donglene til serveren. Brukeren kunne så velge hvilke som skulle tilkobles, i dette tilfellet de

to første i listen. Switch-kommandoen brukes til å fortelle hvilken dongel den påfølgende dataen skal sendes til. Her sendes CON-kommandoen til de to robotene som har blitt koblet til.

```
*scan
*stop
*list
*conn 0
*conn 1
*switch 0
{S, CON}\n
*switch 1
{S, CON}\n
```

Meldingene som ble sendt var alle ASCII-enkodet tekst. Et eksempel er CON kommandoen som ble brukt i eksempelet over. Meldingene startet og sluttet med krøllparentes, og til slutt ny-linje tegnet '\n'. Dette tegnet ble brukt som skilletegn mellom meldinger, når dette ble avlest visste mottakeren at en komplett melding var mottatt.

Når server-dongelen mottok data fra en robot-dongel, la den til ID og navnet på roboten i meldingen før dataen ble videresendt. For eksempel dersom NXT-roboten sendte en idle-melding, ville den sende "{S,IDL}", og det som kom frem hos serveren etter å ha blitt behandlet i server-dongelen var "[0]:nxt:{S,IDL}\n". Basert på det innledende tallet og navnet visste server-applikasjonen hvem som hadde sendt dataen.

8.2 Ny løsning

Blant målene med overhalingen var å implementere en pålitelig kommunikasjon med feilsjekk og retransmisjon. Med andre ord få en forsikring om at den dataen som sendes faktisk kommer fram til mottakeren, og at dataen ikke har blitt korrumpert på veien. I tillegg var det ønsket å få kommunikasjonen mellom robotene og serveren til å ligne mer på et nettverk der donglene oppfører seg som rutere, og pakkesvitsjing tas i bruk for å sende data fra avsender til mottaker.

Pakkesvitsjing vil si at en melding deles opp i mindre biter, der hver bit blir tillagt adressen til mottakeren, slik at nodene i nettverket kan videresende biten til riktig sted.

For å komme nærmere et pakkesvitsjet nettverk, var det ønskelig å fjerne hele kommandosystemet, og formatere all data som sendes i pakker med informasjon om avsender og mottaker. Sentraldonglen skulle heller ikke lenger ha en spesialrolle slik den hadde tidligere med kommandosystemet, men ha samme funksjon som de andre: ta inn en datapakke og sende den videre basert på en mottakeradressen i pakken.

Å enkode pakkene med ASCII ville være lite hensynsmessig, og derfor ble det bestemt at dataen som overføres skulle være rå bytes.

Med disse prinsippene i bunn var det ønsket å implementere kommunikasjonen i lag inspirert av OSI-modellen. Dette er fordelaktig dersom systemet endres på noen måter i fremtiden, for da kan bare de påvirkede lagene byttes, mens de andre beholdes.

8.2.1 Automatisk tilkobling av dongler

Kommandoene som ble brukt tidligere ble blant annet brukt til å be sentraldongelen om å skanne etter og koble til robot-dongler. Når kommandoene ikke lenger skulle brukes, måtte det implementeres en ny prosedyre for tilkobling av dongler.

Uten kommandoene er skanningen og tilkoblingen av periferier noe sentralen må styre selv. Den er derfor blitt satt opp til å skanne hele tiden, og dersom en advertisement blir mottatt fra en enhet som blir identifisert som en robot-dongel, blir denne koblet til automatisk. Dette gjør at roboter kan skrus av og på når som helst, og donglene blir automatisk til- og frakoblet server-dongelen.

En utfordring med kontinuerlig skanning er at skanning benytter seg av Bluetooth-radioen. Altså kan ikke radioen brukes til andre ting samtidig, som å overføre data med en allerede tilkoblet enhet. Skanningen kan tilpasses med hensyn på hvor ofte den skanner, og hvor lenge den lytter ved hver skanning. Jo oftere og lengre det skannes, jo mer er radioen opptatt og forsinker overførselen av data.

På den annen side vil lite skanning gjøre at det tar lang tid å oppdage en periferi. For at en periferi skal oppdages må den sende ut en annonsering samtidig som radioen hos sentralen skanner. Hvis periferien annonserer sjelden, og sentralen skanner sjelden, er det lite sannsynlig at disse to hendelsene finner sted samtidig. En periferi kan stilles inn til å annonsere ofte, slik at sentralen også ved sporadisk skanning har stor sjanse for å oppdage en annonsering.

En annen faktor her er at annonsering skjer på tre forskjellige frekvenser som det hoppes mellom. Derfor må både annonsering og skanning ikke bare skje samtidig, men også på samme frekvens.

På bakgrunn av disse betingelsene ble følgende verdier brukt:
- Sentral skanner hvert 100 ms
- Sentral skanner 6,25 ms hver gang
- Periferi annonserer hvert 40 ms

Dette gjør at radioen er opptatt med skanning i 6,25 ms hvert 100 ms, og har således 93,75 ms til å gjøre andre ting. Med disse verdiene tar det mellom ett og tre sekunder etter at roboten blir skrudd på før annonsering og skanning sammenfaller og en tilkobling opprettes. Dette ble sett på som en akseptabel forsinkelse. Det ble heller aldri observert noen problemer med forsinkelse av data grunnet overdreven skanning.

8.2.2 Nettverkslag

Resultatet av at donglene nå sammenkobles automatisk, er at rett etter at en robot er skrudd på er den tilkoblet et nettverk sammen med serveren og de andre robotene. Selv om det finnes en link mellom alle enhetene i systemet, må det også spesifiseres hva som skal overføres på denne linken, og hvordan de forskjellige enhetene skal behandle dataen som overføres.

Spesifikasjonen på dette er sammenfattet og implementert i et lag som er kalt *nettverkslaget*. Dette laget har ansvar for å ta imot data fra høyere lag, putte dataen i rammer med formatet i figur 8.1, og sende dette på nettverket til riktig mottaker. All data som sendes i systemet blir sendt som en del av datafeltet i en slik ramme.

Figur 8.1: Nettverksramme

Mottaker- og avsenderfeltene er på én byte hver og inneholder adressen til de respektive enhetene. Adresser defineres som konstanter i nettverkslaget i hver robot og på serveren. Altså er det viktig å passe på at alle blir gitt unike adresser. Å implementere en form for dynamisk adresseallokering hadde vært bekvemt, men er ikke gjort.

Protokollfeltet er også på én byte og spesifiserer hvilken protokoll dataen i rammen hører til. Hvilke protokoller som finnes beskrives i neste avsnitt. Datafeltet inneholder de bytene som

skal overføres fra avsender til mottaker.

Til slutt kommer en 8-bit CRC sjekksum. Sjekksummen er en verdi som blir beregnet fra alle de foregående bytene i rammen, og sendes sammen med dem slik at mottakeren kan utføre den samme utregningen, og ved sammenligning finne ut om dataen har blitt korrumpert på veien. Algoritmen er den samme som for protokollen mellom NXT og IO-mikrokontrolleren [2]. Siden sjekksummen er på 8 bit betyr det at alle tenkelige meldinger vil få én av 256 mulige sjekksummer. Det er derfor mulig at riktig sjekksum blir beregnet fra en korrumpert melding. Ved å benytte en sjekksum med 16 eller 32 bit kunne sannsynligheten for at dette skjer minskes betraktelig, men 8 bit ble sett på som akseptabelt i dette systemet.

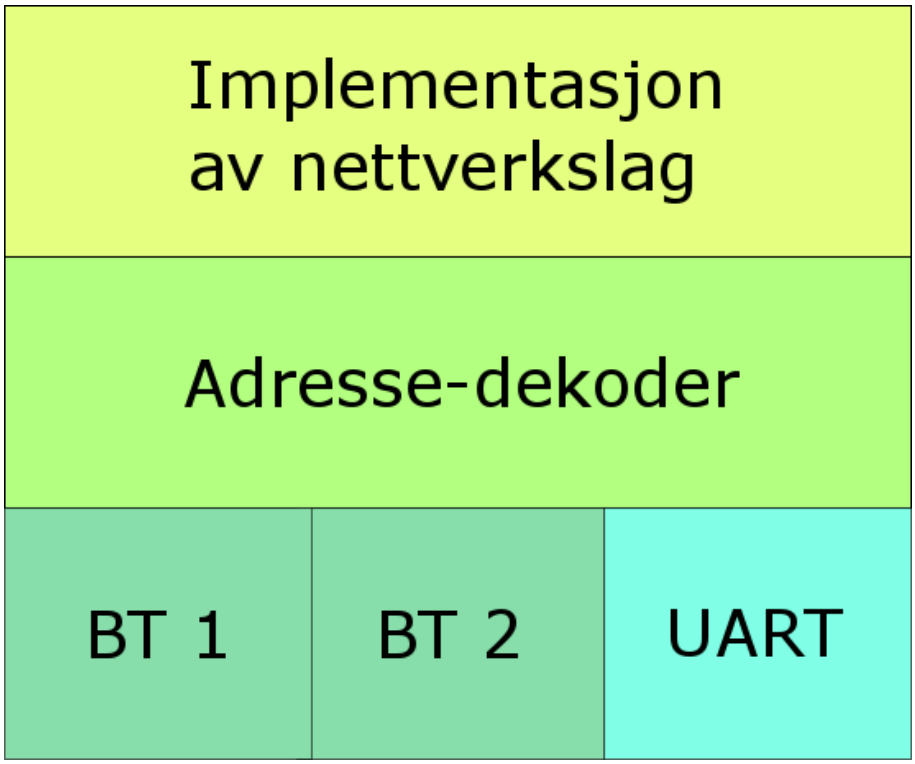
Rammene har variabel lengde og inneholder ikke noe felt som angir hvor mange byte den aktuelle rammen inneholder. Av den grunn må det noe til for å fortelle mottakeren når en komplett ramme er tatt imot. I det gamle systemet var det ”ny linje”-tegnet som signaliserte dette. Å skille meldinger er enkelt i en tekstbasert protokoll fordi ett tegn kan settes av til å markere dette, for så å passe på å aldri sende tegnet som en del av en melding. Det er ikke like enkelt når det er rå bytes som sendes.

For å løse dette enkodes alle nettverksrammene med COBS før de sendes. Se avsnitt 6.7.2 for mer informasjon om COBS.

Dongler

Donglene opererer på nettverkslaget, og må inneholde funksjonalitet for å håndtere nettverksrammene. I realiteten, slik Bluetooth-topologien er satt opp i dag som en stjerne med sentraldongelen i midten, er det kun midtpunktet som behøver en implementasjon av nettverkslaget. De andre trenger kun videresende data mottatt på UART over Bluetooth, og motsatt.

Dongelen i midten derimot må implementere funksjonalitet fra nettverkslaget for å bestemme hvor pakkene skal sendes. Dongelen har to typer tilkoblinger: UART og Bluetooth. Det finnes én UART-tilkobling, til datamaskinen som kjører serveren, og én Bluetooth-tilkobling for hver påskrudde robot. For å abstrahere vekk at data kommer fra forskjellige fysiske tilkoblinger, er det laget en adresse-dekoder modul som oversetter mellom protokoll-adresser og fysisk tilkoblingspunkt. Selve implementasjonen av nettverkslaget operer derfor bare med adresser og det er usynlig for denne at noe data kommer fra UART og noe fra Bluetooth-tilkoblede enheter. Figur 8.2 illustrerer prinsippet.



Figur 8.2: Lag i sentral-dongel

I illustrasjonen er det to tilkoblede Bluetooth-enheter i tillegg til UART-tilkoblingen. Når data-bytes blir mottatt fra en av de fysiske tilkoblingene blir disse gitt til adressedekoderen. Denne leverer dataen videre til nettverkslaget. Når en av de innkommende bytene er '0' signaliserer dette at en komplett nettverksramme er mottatt. Denne blir COBS-dekodet og en CRC-sjekk utføres. Avsender- og mottakeradresse blir hentet ut av rammen, og gitt til adresse-dekoderen sammen med de mottate bytene. Dekoderen inneholder en tabell som knytter sammen tilkobling og nettverks-adresse. Denne slår dekoderen opp i og prøver først å finne avsenderadressen. Dersom ingen fysisk tilkobling er lagret for avsenderen, blir dette gjort nå, slik at når noe data i fremtiden kommer med denne adressen som mottaker, vet dekoderen hvor dataen skal sendes. Deretter slår dekoderen opp mottaker-adressen og henter ut hvilken tilkobling mottakeren kan nås på og sender dataen dit. Dersom dekoderen ikke vet hvor mottakeren befinner seg blir rammen kringkastet på alle tilkoblingene. Dette kan den gjøre fordi nettverkslagene i de enkelte enhetene sjekker om mottatte rammer er adressert til seg selv, og hvis det ikke er tilfellet blir rammen kastet.

8.2.3 Transportlag

Over nettverkslaget måtte det implementeres et lag som tok i bruk funksjonaliteten i nettverkslaget for å overføre meldinger. Dette laget skulle ha ansvaret for å dele en melding med

en hvilken som helst størrelse i mindre biter, dersom det er nødvendig, og sende disse gjennom nettverkslaget. For å tilfredsstille ønsket om å ha en pålitelig kommunikasjon måtte dette laget definere en protokoll som sender bekreftelser når data mottas, og sender dataen på nytt dersom en bekreftelse ikke kommer. For å oppnå større fleksibilitet var det også ønsket en ikke-pålitelig protokoll på dette laget, slik at data det ikke er så farlig om forsvinner kan sendes med mindre overhead.

Pålitelig protokoll

En pålitelig protokoll sender data på nytt dersom mottakeren ikke bekrefter at dataen er kommet frem. På Internett er det TCP-protokollen som står for denne funksjonaliteten. Protokollen som er utviklet og implementert for robot-systemet henter inspirasjon fra TCP.

Denne metoden for dataoverføring kalles automatic repeat request (ARQ). Basert på dette prinsippet kan det utvikles protokoller med forskjellig virkemåte og forskjellig grad av kompleksitet.

En enkel versjon av ARQ er en implementasjon der avsenderen må vente på bekreftelse etter hver sendte melding. Etter at dataen er sendt, får ikke avsenderen sende noe mer før en bekreftelse er mottatt. Når bekreftelsen kommer blir neste bolk med data sendt. Et resultat av at det bare er lov å sende én melding før man må stoppe og vente, er at mye tid kastes bort på venting istedenfor å overføre data.

En videreutvikling for å forbedre dette er å tillate senderen å ha et gitt antall meldinger utestående som ikke har blitt bekreftet. Hvis dette tallet for eksempel settes til fem, kan fem meldinger sendes uten at noen bekreftelse er mottatt for dem. De utestående meldingene lagres i det som kalles transmisjonsvinduet, og størrelsen på vinduet bestemmer hvor mange som kan sendes uten bekreftelse. Når fem meldinger har blitt sendt er avsenderen nødt til å vente på at minst én bekreftelse kommer slik at det blir frigjort plass i transmisjonsvinduet, før noe mer kan sendes.

Det finnes to distinkte varianter av vindusprotokollen, og forskjellen på disse er om mottakeren kan motta meldinger i feil rekkefølge. Dersom dette er tillatt og en melding går tapt, vil mottakeren likevel ta imot og bekrefte meldingene som kommer etter den tapte, men samtidig si ifra at det er en melding den ikke har fått. Når den tapte meldingen sendes på nytt blir de mottatte meldingene satt sammen i riktig rekkefølge. I det andre tilfellet vil mottakeren nekte å motta noe som ikke har det sekvensnummeret som er én større enn den forrige

den mottok. Når en melding da går tapt og de påfølgende meldingene blir mottatt, blir disse kastet og en bekreftelse blir sendt med det sekvensnummeret den forventer å motta. Når avsenderen etter en tid oppdager at den har fått noen bekreftelse enda, blir alle meldingene lagret i transmisjonsvinduet sendt på nytt.

Hvilken metode som skulle velges for robotkommunikasjonen måtte bestemmes basert på kravene til systemet. Metodene øker i kompleksitet, men også i effektivitet og utnyttelse av kommunikasjonslinken. For systemet måtte det avgjøre hva som skulle vektlegges.

Det ble avgjort at kravene til effektivitet ikke er så fryktelig store i dette systemet, da mengden data som overføres ikke er for høy. Likevel kunne det være fordelaktig å ha mulighet til å sende flere meldinger på en gang uten å måtte vente. Basert på dette falt valget på metoden som tillater flere sendte meldinger, men der mottakelse må skje i riktig rekkefølge.

Implementasjonen av protokollen basert på denne metoden definerer meldingen i figur 8.3. For å følge terminologien i TCP-protokollen kalles her denne dataenheten for et *segment*.

Figur 8.3: ARQ-segment

Typefeltet inneholder hva slags ARQ-segment dette er, og kan inneholde typene i listen under.

1. Data
2. Bekreftelse
3. Synkronisering
4. Bekreftelse på synkronisering
5. Lever-test

Et data-segment er den eneste av typene som inneholder data. Når et segment med denne typen blir mottatt, sendes et segment med type 2 (bekreftelse) til avsenderen. For hver datamelding som sendes øker sekvensnummeret med én. Sekvensnummeret i et bekreftelsesegment er lik nummeret på det segmentet mottakeren forventer å motta neste gang.

Protokollen bruker sekvensnummer fra 0 til 127, deretter starter tellingen på 0 igjen. Dette betyr at vindusstørrelsen heller ikke kan være større enn 127.

Siden protokollen er pålitelig er det ikke noen vits å prøve å sende data til en mottaker det ikke vites om eksisterer. Derfor settes en forbindelse opp mellom mottaker og avsender før noe data utveksles. En server som det er mulig å koble til må kontinuerlig lytte etter innkommende segmenter med synkroniseringstypen, og svarer på en slik med bekreftelse på synkronisering. Når klienten mottar denne bekreftelsen er forbindelsen satt opp og data kan utveksles. Etter at forbindelsen er satt opp vil lever-du-tester bli sendt av serveren med jevne mellomrom som klienten bekrefter, for å forsikre at forbindelsen ikke har gått tapt.

Det er ikke noe feilsjekk av ARQ-segmentene, dette håndteres bare i nettverkslaget. Dersom nettverkslaget mottar en korrumpert melding, blir denne kastet. Hvis dette var et ARQ-segment vil det utløse en retransmisjon.

Når et segment skal sendes gis det til nettverkslaget, der hele segmentet ender opp som datafeltet i en nettverksramme. Protokoll-feltet i nettverksrammen vil bli satt til å signalisere at innholdet er et ARQ-segment. Se figur 8.1.

Det ønskes en kjapp meldingsoverføring og hurtig respons fra begge partene slik at en bekreftelse kommer hurtig etter at meldinger er sendt. Avsenderen må sette en frist for når den skal sende segmenter på nytt som det ikke er mottatt bekreftelse for. Dersom denne fristen er for stor vil det påvirke kommunikasjonen negativt da det vil da lang tid før tapte meldinger sendes på nytt. Det ønskes jo et så kjapt og responsivt system som mulig. Det er derfor fordelaktig å ha så lav frist som mulig, men som likevel er høy nok til at mottakeren rekker å sende en bekreftelse. For å oppnå dette er det ønskelig å minimere forsinkelser som finnes rundt om i systemet. Slike forsinkelser er for eksempel på grunn av frekvens på tasker som leser innkommende meldinger og donglers kommunikasjonsintervall. På NXT innfører også IO-mikrokontrolleren en ekstra forsinkelse.

Forsinkelsen mellom melding og bekreftelse ble testet ved å sende en lever-du-test hvert femte sekund til en robot og måle tiden til bekreftelse ble mottatt. Uten noen forbedringer utgjorde dette totalt mellom 100 og 300 ms. I et forsøk på å få dette tallet ned ble det gjort noen endringer. Tasken som leser innkommende meldinger var til å begynne satt til å sove 10 ms mellom hver gang innboksen ble sjekket. Denne soveperioden ble fjernet helt. En endring som sannsynligvis hadde mer innvirkning ble gjort i donglene. Når en Bluetooth-tilkobling settes opp mellom to enheter kan det bestemmes hvor ofte sentralen skal be periferien om data. Det settes en øvre og nedre grense for hvor langt intervallet kan være, og så avgjør sentralen fra gang til gang, basert på hva som passer best for den, når den ber om data i dette intervallet. I utgangspunktet var dette satt til mellom 75 og 20 ms. Ved å sette begge deler til 7.5 ms, som er det minste tillatte, vil overføringshastigheten øke. Resultatet var at tiden fra

sendt melding til mottatt bekreftelse lå på mellom 30 og 120 ms.

Simpel protokoll

Den pålitelige protokollen beskrevet i forrige avsnitt er en slektning av TCP i Internets protokollkatalog. Denne katalogen inneholder også UDP-protokollen som i motsetning til TCP er forbindelsesløs, og sender data uten å vite om mottakeren eksisterer og uten å bry seg om dataen kommer frem. Å sende data slik er nyttig i mange sammenhenger, og det var derfor ønskelig å ha noe lignende i robotsystemet.

En simpel protokoll er derfor implementert som muliggjør dette. I UDP kalles dataenhetene for datagram, og denne navnekonvensjonen brukes også her. Formatet på datagrammene i den simple protokollen er vist i figur 8.4.

Figur 8.4: Simpel-datagram

Når en melding sendes med den simple protokollen deles den i mindre biter dersom den er lengre enn grensen satt av nettverkslaget. Nummer-feltet i datagrammet inneholder hvilket nummer den aktuelle biten har i rekkefølgen av mindre biter. Siste nummer inneholder nummeret på den siste biten av meldingen. Det vil si at en melding som det er plass til i ett datagram vil ha nummer lik ’0’ og siste nummer lik ’0’. Dersom meldingen må deles i to biter vil den første ha 0/1 og den siste 1/1.

Disse nummerene blir brukt av mottakeren for å sette sammen meldingen igjen. Dersom et datagram går tapt eller blir kastet av nettverkslaget i feilsjekken, oppdager mottakeren det når nummerene ikke kommer i den riktige rekkefølgen. I det tilfellet blir hele meldingen kastet og det blir ikke gjort noe forsøk på å rette opp i feilen.

Datagrammene gis til nettverkslaget og ender opp i datafeltet i en nettverksramme, med protokoll-feltet satt til å vise at innholdet skal håndteres av den simple-protokollen hos mottakeren.

8.2.4 Meldingsformat

Meldingene som sirkuleres i systemet, slik som oppdatering fra robot om målinger eller kommando fra server om hvor roboten skal bevege seg, sendes gjennom transport- og nettverkslagene.

Tidligere var disse meldingene formatert som tekst, eksempelvis kommandoer fra serveren hadde formatet ”{U,90,100}\n”. Akkurat denne meldingen ville bedt roboten om å rotere 90° og kjøre 100 cm fremover.

Meldingene kunne helt fint sett slik ut også med den nye kommunikasjonen, teksten ville da havnet i datafeltet til et ARQ-segment eller simpel-datagram. Dog som nevnt tidligere er det lite effektivt å sende numeriske verdier som tekst, i tillegg til at det er litt pussig å gå den omveien. Meldingen over ville krevd én byte for hvert tegn: totalt 11 byte. Ved å heller sende numeriske verdier ville den samme meldingen krevd 5 byte: én for meldingstype og to hver for vinkel og avstand. Med to byte kan verdier opp til 65535 sendes. Med én byte ville maks vært 255, og rotasjoner opp til 360° eller avstander over 255 cm ville ikke vært mulig.

Den samme meldingen sendt som numeriske verdier ville sett slik ut, representert heksadesimalt:

0x01	0x05A 0x00	0x64 0x00
Type	Vinkel (90)	Avstand (100)

For å slippe å konvertere til og fra tekst, og å sende unødvendig mange bytes, er alle meldingene i den nye løsningen laget på et format der innholdet er rene tall. En oversikt over hvordan de nye meldingene er formatert finnes i vedlegg A. En oppsummering over forskjellen i lengde på de nye og gamle meldingene er vist i tabell 8.1.

Type	Ny	Gammel
Handshake	26-35	49
Order	5	11
Update	13	32
Idle, pause, unpause, connect	2	8

Tabell 8.1: Meldingsstørrelser før/etter

Siden antall byte varierer når meldingene sendes som tekst, for eksempel vinkelen 360 bruker 3 byte mens 90 krever 2, er ikke verdiene for det gamle systemet i tabellen absolutt. Antallet som er brukt i tabellen er middelverdier.

Her er det ikke tatt hensyn til den ekstra overhead meldingene vil få grunnet ekstra informasjon som legges til i transport- og nettverkslagene. Dette er uansett irrelevant når man sammenligner meldingene opp mot hverandre, da disse ekstra bytene ville forekommet også hvis tekst-meldingene skulle sendes med det forbedrede kommunikasjonssystemet. I det gamle systemet var det også betydelig overhead når server-dongelen legger til "[0]:NXT:"på alle meldingene den videresender til server-applikasjonen. I tillegg sendte server-dongelen tidligere bare 20 byte av gangen, derfor ville handshake-meldingen blitt sendt i tre omganger, med ekstra overhead alle gangen. Dette resulterte i at antall byte sendt til serveren var over 80.

Basert på denne informasjonen kan man si at byte-mengden som sendes nå er mindre enn før. I tillegg til at behandlingen av meldingene er forenklet.

Viktig å nevne er også at kommunikasjonen bruker little-endian rekkefølge på bytene. Det vil si at dersom en verdi i meldingen består av mer enn én byte, sendes disse med rekkefølge fra minst til mest signifikant. For eksempel vinkelen 270, heksadesimalt 0x010E, sendes som 0x0E 0x01.

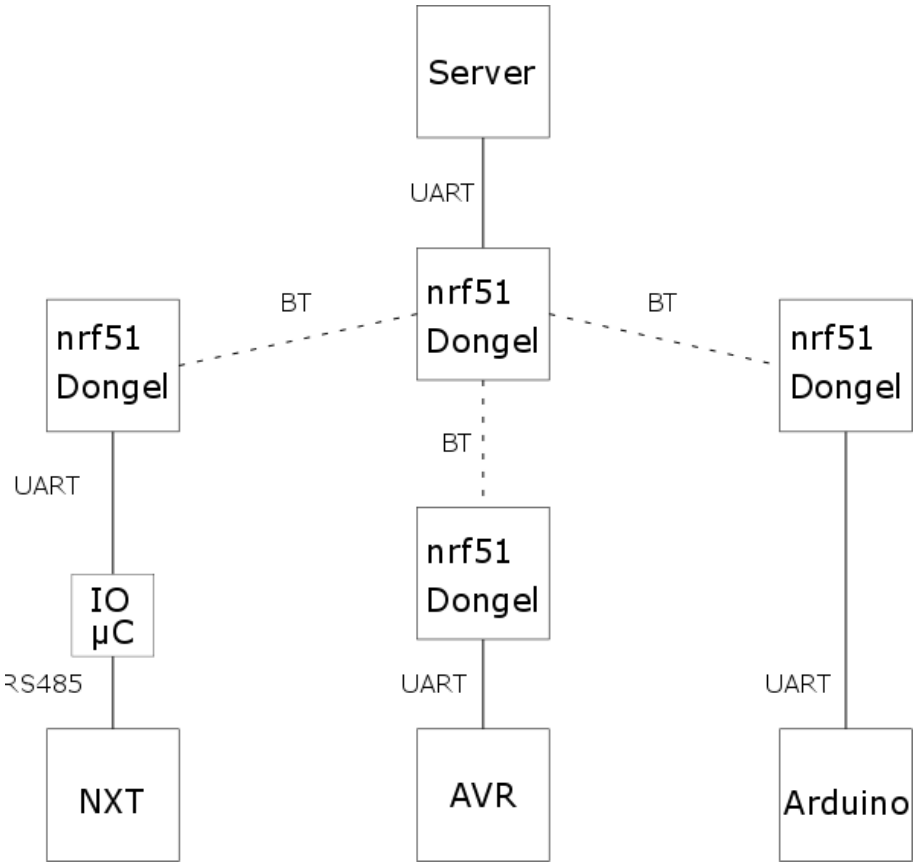
8.3 Resultat

Kommunikasjonssystemet beskrevet i dette kapittelet er implementert på NXT-, Arduino- og AVR-roboten. I tillegg er kommunikasjonsmodulen i serverapplikasjonen skrevet på nytt for å ta bruk samme funksjonalitet.

De forskjellige lagene definerer forskjellige innstillinger som kan modifiseres for å oppnå ønsket virkemåte. For eksempel kan det enkelt på applikasjonsnivå endres hvilke meldingstyper som bruker hvilken transport-protokoll. Per nå er disse satt slik at alle meldingene untatt oppdateringsmeldinger fra robotene bruker ARQ. Dette betyr at alle kommandoer og informasjon om status blir sendt pålitelig og er garantert å komme frem. Oppdateringsmeldingene sendes hvert 200 ms, og dersom en av disse går tapt har ikke dette egentlig noe å si for systemet. Derfor sendes disse med den simple protkollen for å ikke legge like stort press på retransmisjons-protokollen.

Videre kan det endres hvor store nettverksrammene kan være, hvor stort ARQ transmisjonsvinduet er og maks størrelse på applikasjonsmeldingene. Ved større verdier på disse blir mer minne allokert til buffere. Å ha mye større buffere enn applikasjonen bruker av trafikk er

sløsing, men for små kan føre til overflow på buffere.



Figur 8.5: Kommunikasjonen i systemet

Bluetooth nettveket er fortsatt satt opp i sjernetopologi som er den enkleste måten, men mer avanserte topologier er mulig å sette opp med Bluetooth Low Energy. Protokollene som er laget er designet for å passe i alle pakkesvitsjede nettverk, så dersom for eksempel en mesh topologi skulle bli implementert i fremtiden er systemet klar for dette. Det som da må endres er at periferidonglene må settes opp med samme implementasjon av nettverkslaget som sentral-dongelen.

I tillegg måtte implementasjonen blitt utvidet med logikk for å finne mest effektive veg til mottaker. Nå er det alltid bare én vei til målet.

Tabell 8.2 viser en sending av handshake-melding med ARQ-protokollen. I dette eksempelet er maks størrelse på nettverksrammene satt kunstig lavt (10 bytes) for å vise virkemåten. Transport- og nettverkslaget bruker 8 byte, og derfor er det bare to bytes i hver ramme som brukes til nyttelast. Handshaken er på 28 bytes og altså trengs 14 rammer for å overføre hele handshaken. Det er sendingen av disse 14 rammene som er vist i tabell 8.2. Loggingen av informasjonen er gjort på sentral-dongelen. Denne sniffer pakkene som sirkuleres og printer innholdet. Sever-dongelen er for eksempelets skyld satt opp til å ha 10 % sannsynlighet for å

ikke videresende en ramme, for å simulere tapte pakker. Rammene som er truffet og ikke blir videresendt er markert med rødt. Tabellen viser tid i millisekunder og adressene til avsender og mottaker. Roboten har her adresse 1, og meldingene den sender har grå bakgrunnsfarge. Serveren har adresse 0. Rett før denne sekvensen har det blitt utvekslet et synkroniserings-håndtrykk som har satt opp en forbindelse mellom enhetene.

Status	Tid	Fra	Til	Type	Sekvensnummer	Antall byte
OK	4487	1	0	Data	0	10
OK	4496	0	1	Bekreftelse	1	8
Tapt	4506	1	0	Data	1	10
OK	4526	1	0	Data	2	10
OK	4534	0	1	Bekreftelse	1	8
OK	4546	1	0	Data	3	10
OK	4554	0	1	Bekreftelse	1	8
OK	4566	1	0	Data	4	10
OK	4575	0	1	Bekreftelse	1	8
OK	4745	1	0	Data	1	10
OK	4754	0	1	Bekreftelse	2	8
OK	4764	1	0	Data	2	10
OK	4774	0	1	Bekreftelse	3	8
OK	4784	1	0	Data	3	10
OK	4793	0	1	Bekreftelse	4	8
OK	4804	1	0	Data	4	10
OK	4814	0	1	Bekreftelse	5	8
OK	4824	1	0	Data	5	10
OK	4832	0	1	Bekreftelse	6	8
OK	4844	1	0	Data	6	10
OK	4853	0	1	Bekreftelse	7	8
OK	4864	1	0	Data	7	10
Tapt	4873	0	1	Bekreftelse	8	8
OK	4884	1	0	Data	8	10
OK	4892	0	1	Bekreftelse	9	8
OK	4903	1	0	Data	9	10
OK	4912	0	1	Bekreftelse	10	8
OK	4943	1	0	Data	10	10
OK	4952	0	1	Bekreftelse	11	8
OK	4963	1	0	Data	11	10
OK	4972	0	1	Bekreftelse	12	8
OK	4983	1	0	Data	12	10
OK	4993	0	1	Bekreftelse	13	8
OK	5003	1	0	Data	13	10
Tapt	5015	0	1	Bekreftelse	14	8
OK	5241	1	0	Data	13	10
OK	5250	0	1	Bekreftelse	14	8

Tabell 8.2: Overføring av handshake med ARQ, maks 10 byte ramme

Eksempelet demonstrerer et par interessante og karakteristiske trekk ved protokollen:
1. Når roboten sender data med sekvensnummer 1 går denne tapt. Roboten fortsetter å sende så mange pakker som det er plass til i transmisjonsvinduet, mens serveren svarer på hver av disse med bekreftelse på sekvensnummer 1, som er den pakken den forventer å motta men som har gått tapt. Vinduet har her størrelse 4, så roboten fyller opp vinduet og kan deretter ikke sende mer. Når fristen for bekreftelsen går ut 200 ms senere trigges en retransmisjon av alle pakkene i vinduet. Alle disse sendes feilfritt og roboten fortsetter å sende sekvensnummer 5.

2. Data fra robot med sekvensnummer 7 blir sendt vellykket, men bekreftelsen fra server på denne går tapt. Roboten fortsetter å sende neste pakke da det er plass i vinduet. Siden den neste pakken er den serveren forventer å motta, sender den derfor en bekreftelse. Når roboten mottar denne har den mottatt bekreftelse på at serveren forventer å motta 9 neste gang. Altså vet roboten at alt frem til sekvensnummer 8 har kommet frem, på tross av at en tidligere bekreftelse fra server gikk tapt. Følgelig blir det ingen retransmisjon av ramme 7 som roboten aldri har mottatt bekreftelse på. Dette er en av fordelen med denne versjonen av ARQ der roboten kan ha utestående meldinger som ikke er bekreftet. Dersom roboten måtte stoppet og vente på bekreftelse etter hver melding, ville det her oppstått en unødvendig retransmisjon.

3. Data med sekvensnummer 14 blir sendt feilfritt, men igjen går bekreftelsen fra server tapt. Forskjellen fra forrige punkt er at dette var den siste rammen som skulle sendes. Altså kommer det ikke flere rammer som serveren har mulighet til å sende en bekreftelse på og likevel få sagt ifra at den har mottatt alt. Når fristen går ut oppdager roboten at den har én utestående pakke som ikke har blitt bekreftet. Denne blir sendt på nytt, serveren mottar denne, og sender så en bekreftelse som kommer frem og begge parter vet at handshaken har blitt overført korrekt. Når serveren mottar den siste pakken på nytt, ser den av sekvensnummeret at dette er noe den har mottatt tidligere, så den vil bli ignorert, men sender likevel en bekreftelse da det må ha forekommet en feil ved forrige sending.

Tabell 8.3 viser logging av rammer sendt med både simpel- og ARQ-protokoll. Meldingene sendt med simpel er oppdateringer fra robot hvert 200 ms om målinger den gjør. ARQ-meldingene er lever-du-test fra server. Det sees at oppdateringsmeldingene som går tapt ikke utløser noen reaksjon, men når bekreftelsen på lever-du-meldingen med sekvensnummer 5 går tapt trigges en retransmisjon.

Status	Tid	Fra	Til	Type	Sekvensnummer	Antall byte
Tapt	912	1	0	Simpel	-	21
OK	1110	1	0	Simpel	-	21
Tapt	1309	1	0	Simpel	-	21
OK	1508	1	0	Simpel	-	21
Tapt	1706	1	0	Simpel	-	21
OK	1905	1	0	Simpel	-	21
OK	2103	1	0	Simpel	-	21
OK	2215	0	1	Data	5	10
Tapt	2262	1	0	Bekreftelse	6	8
OK	2305	1	0	Simpel	-	10
OK	2414	0	1	Data	5	8
OK	2461	1	0	Bekreftelse	6	8
OK	2501	1	0	Simpel	-	21
OK	2699	1	0	Simpel	-	21
OK	2898	1	0	Simpel	-	21

Tabell 8.3: Forskjellig virkemåte mellom ARQ og simpel protokoll

8.3.1 Tester

For å teste kapasiteten og mulighetene ved den nye kommunikasjonen, har det blitt utført et par tester.

Meldingstap

Én av testene gikk ut på hvor stort meldingstap systemet tåler. Dersom det forekommer et stort antall meldingstap vil tilkoblingen mellom enhetene brytes når en av partene ikke får kontakt med den andre innen en bestemt tid. Akkurat hvor lang tid det skal gå før dette skjer finnes det en innstilling for, og kan endres for å finstille systemet. En annen konsekvens av mye meldingstap er at bufferene som inneholder dataen som skal sendes ved første mulighet, fylles opp. Dersom dette skjer vil forsøk på å sende mer data mislykkes, og i ytterste konsekvens kan enheten med full buffer slutte å fungere og må restartes.

For å teste toleransen for meldingstap ble sentral-dongelen til å begynne med satt til å miste nettverksrammer med en viss sannsynlighet. Dersom denne sannsynligheten inntraff var det deretter 50/50 om meldingen skulle forkastes helt eller videresendes med en endret byte, for å simulere meldingskorrumpering. Det siste ville resultere i at rammen ble kastet av mottakeren når feilsjekken mislykkes. Testene ble kjørt med tilkoblings-timeout dersom kontakt med den andre parten ikke oppnås innen 1000 ms, og retransmisjon dersom bekreftelse ikke

er mottatt innen 200 ms.

1. Med 30% meldingstap og sending av oppdateringsmeldinger med ARQ fylte bufferene seg fort opp og programmet stoppet.
2. Uten ARQ på oppdateringsmeldingene var det fortsatt bare et spørsmål om tid før en melding fra server, eller bekreftelse den andre veien, gikk tapt flere ganger på rad og serveren tolket det som en død robot og timeout på tilkoblingen skjedde.
3. Nede på 20% feilrate kunne systemet kjøre lenge, men etter en stund timer tilkoblingen fortsatt ut når nok lever-du tester på rad går tapt.
4. På 10 % ble det kjørt en test i 10 min der alle tre robotene sto å sendte oppdateringer og ble alive testen uten én feil.

10% tap av meldinger virker å være smertegrensen for hva kommunikasjonen tåler. Dette avhenger riktignok av størrelsen på send-bufferne og hvor lang timeout er satt. Ved å øke begge disse vil også systemet tåle mer feil, men på bekostning av responsivitet og større minnebruk.

Meldingsstørrelse

Videre har det blitt testet hvor store meldinger som kan sendes og hvor lang tid dette tar. Med melding menes her data satt sammen på applikasjons-nivå, og som blir delt i mindre biter av underliggende lag og sendt som deler av nettverksrammer. Testene har benyttet en maks rammestørrelse på 50 byte. Minus protokolloverhead er det da plass til 42 byte med nyttelast i hver ramme.

Den første testen ble gjort med en melding på 1000 byte. Med de innstillingene som var brukt ville dette overføres i 24 rammer. Med 20% tap i sentral-dongelen og sending med ARQ, ble hele meldingen overført korrekt fra robot til server på 6 sekunder. Når feilraten ble senket til 0% tok det samme 1,5 sekunder. Den samme tiden tok det når dataen ble sendt med den simple protokollen.

Når datamengden ble økt til 4000 byte ble også dette overført korrekt. Denne mengden data var på grensen til hvor mye ledig minne roboten hadde og som kunne allokeres til sendbuffere, så tester med større mengder ble ikke gjort.

Disse resultatene viser at det i praksis ikke finnes noen øvre grense hvor store meldingene kan være med den nye kommunikasjonen. Før meldingen blir delt i mindre biter for å sendes, legges to byte til i starten av meldingen og disse inneholder antall byte hele meldingen

består av, slik at mottakeren vet hvor mange biter som må settes sammen igjen for å få tilbake originalmeldingen. Derfor er det i realiteten en øvre grense på 65535 byte, som er den største verdien som kan representeres med 2 byte. Dette er så stort at robotene ikke har nok minne til å definere en melding med slik størrelse. Således begrenses meldingsstørrelsen kun av hvor mye ledig minne robotene har.

Når det kommer til hastighet så utgjør resultatet over uten retransmisjoner i underkant av 700 byte per sekund. Dette kunne sannsynligvis blitt økt ved å bruke større nettverksrammer, slik at dataen blir sendt i færre biter, men da blir det på bekostning av minne, siden større buffere vil trengs. I Bluetooth er det teoretisk mulig å komme opp i 10 kB/s. UART kommunikasjonen mellom roboter og deres dongler kjører på 38400 baud og begrenser derfor maks teoretiske hastighet ytterligere til omtrent 4 kB/s.

Hastigheten som oppnås på nåværende tidspunkt er dog mer enn nok for den mengden data systemet overfører.

Pakkesvitsjing

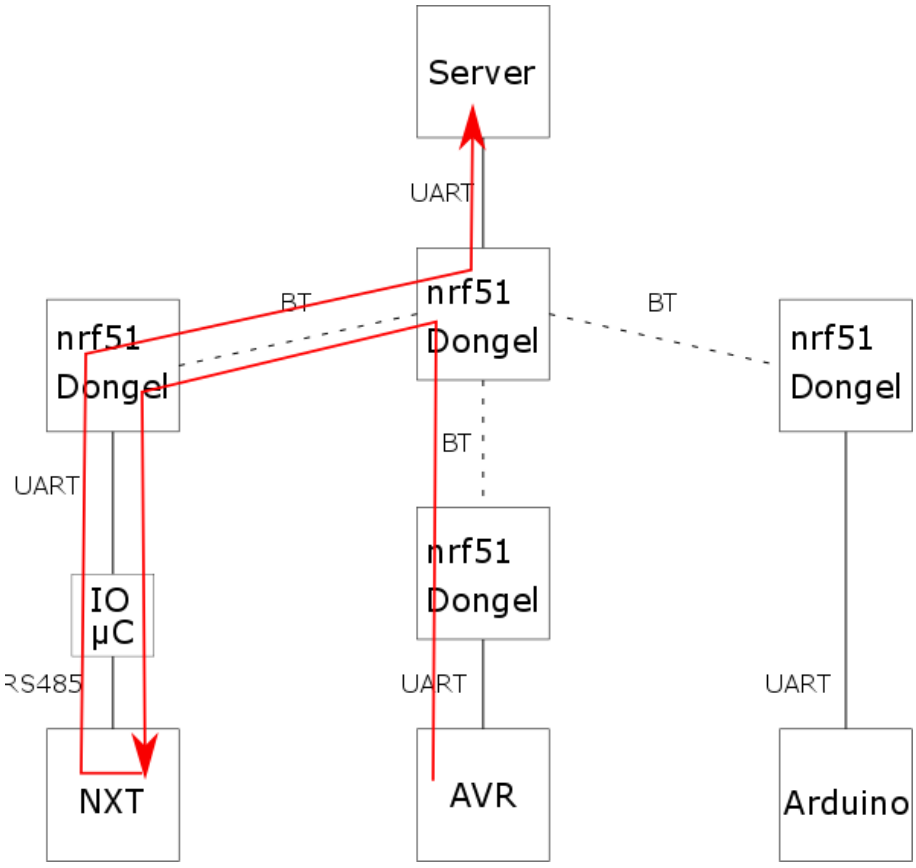
Kommunikasjonen implementerer nå adresser og videresending av rammer basert på disse adressene. Slik systemet er i dag, finnes det ikke noe funksjonalitet som tar i bruk mulighetene dette tilbyr. Nå er det mulig for roboter og sende meldinger direkte til andre roboter, og meldinger kan kringkastes til alle enhetene på nettverket.

Nettverkstopologien er heller ikke optimalisert for slikt bruk. Stjerneformen gjør at uansett hvem dataen kommer fra og hvor den skal må rammene innom sentral-dongelen før de videresendes.

Siden det ikke finnes noen funksjonalitet som tar i bruk mulighetene pakkesvitsjingen tilbyr, ble det satt opp et test-scenario der én av robotene fungerte som et ledd mellom serveren og en annen robot. Se figur 8.6. NXT ble satt opp med en forbindelse til både server og AVR-robot. I AVR-roboten ble server-adressen endret slik at den trodde serveren befant seg på NXT sin adresse. Når AVR ble skrudd på begynte den derfor å sende meldinger til NXT. Disse mottok NXT og sendte videre innholdet til den faktiske server-adressen. Når disse ble mottatt av serveren la den inn AVR-roboten i systemet, men trodde at den befant seg på NXT sin adresse. Alle meldinger fra server til AVR gikk derfor også via NXT-roboten. Oppdateringene fra AVR-roboten kom frem til serveren og ble brukt til å utvikle kartet, og kommandoene fra serveren gjorde at AVR-roboten beveget seg som ønsket.

Dette demonstrerer fleksibiliteten med denne kommunikasjonen: meldinger kan gå mellom hvilke som helst av enhetene som er tilkoblet nettverket. Tidligere var det rigid satt opp at alle meldinger fra en robot går direkte til serveren-datamaskinen.

Dersom nettverkstopolgien i fremtiden skulle nærme seg et mesh-nettverk ville dette sammen med den pakkesvitsjede protokollen gjort kommunikasjonen veldig fleksibel. Eksempelet over kunne da blitt utvidet slik at en robot kunne fungert som mellommann for en annen som er utenfor rekkevidden til server-dongelen.



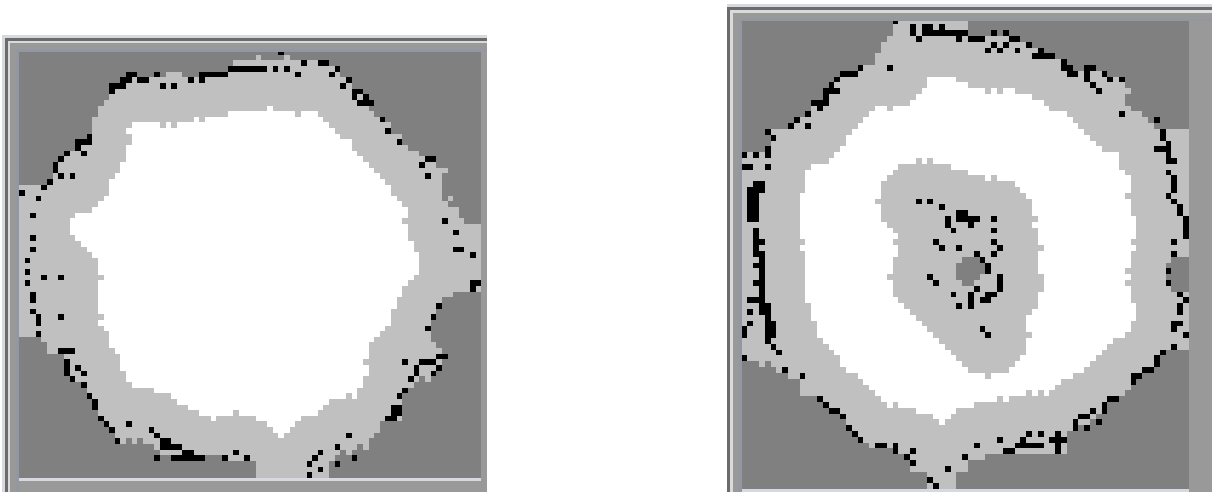
Figur 8.6: Kommunikasjon med NXT som mellomledd

Kapittel 9

Resultater

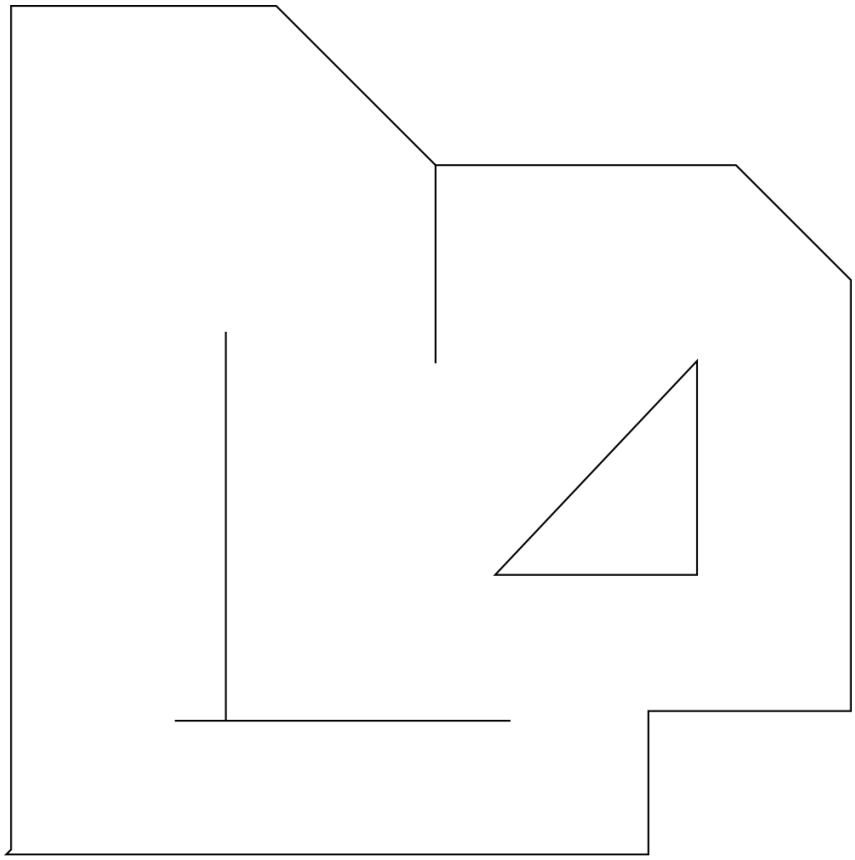
Dette kapittelet vil presentere resultater fra kartlegginger gjort med NXT-roboten etter at all funksjonalitet beskrevet i oppgaven er implementert.

Figur 9.1 viser kart produsert når roboten har kjørt i en sirkulær bane. Til venstre er det ingen ytterligere hindringer bortsett fra veggene, mens bildet til høyre er fra en test der en rektangulær boks ble plassert i sentrum.



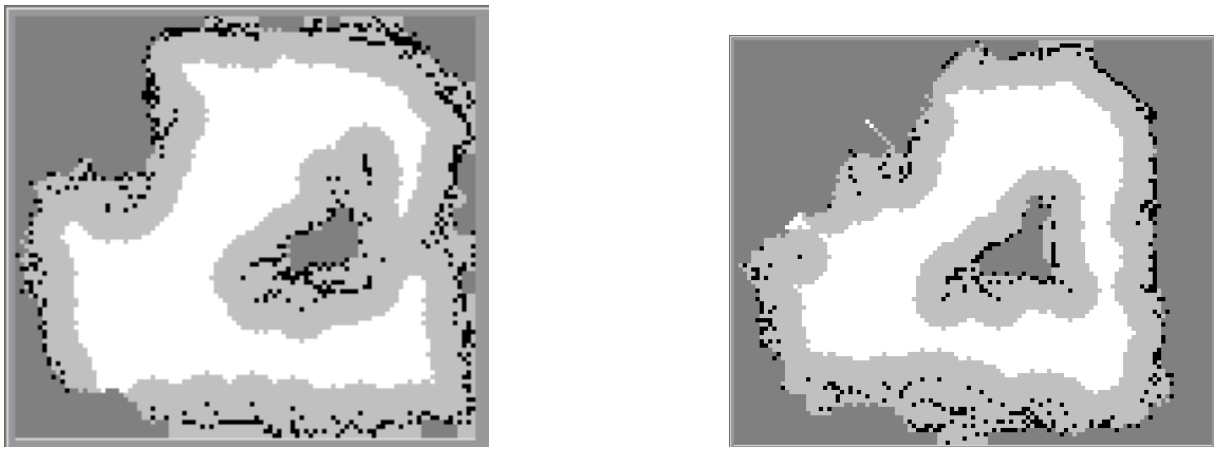
Figur 9.1: Rund testbane

De neste bildene er fra tester gjort i en labyrint med oppsett som skissert i figur 9.2. Under testene har denne blitt avgrenset på forskjellige måter for å skape labyrinter med forskjellige størrelser og kompleksitet.



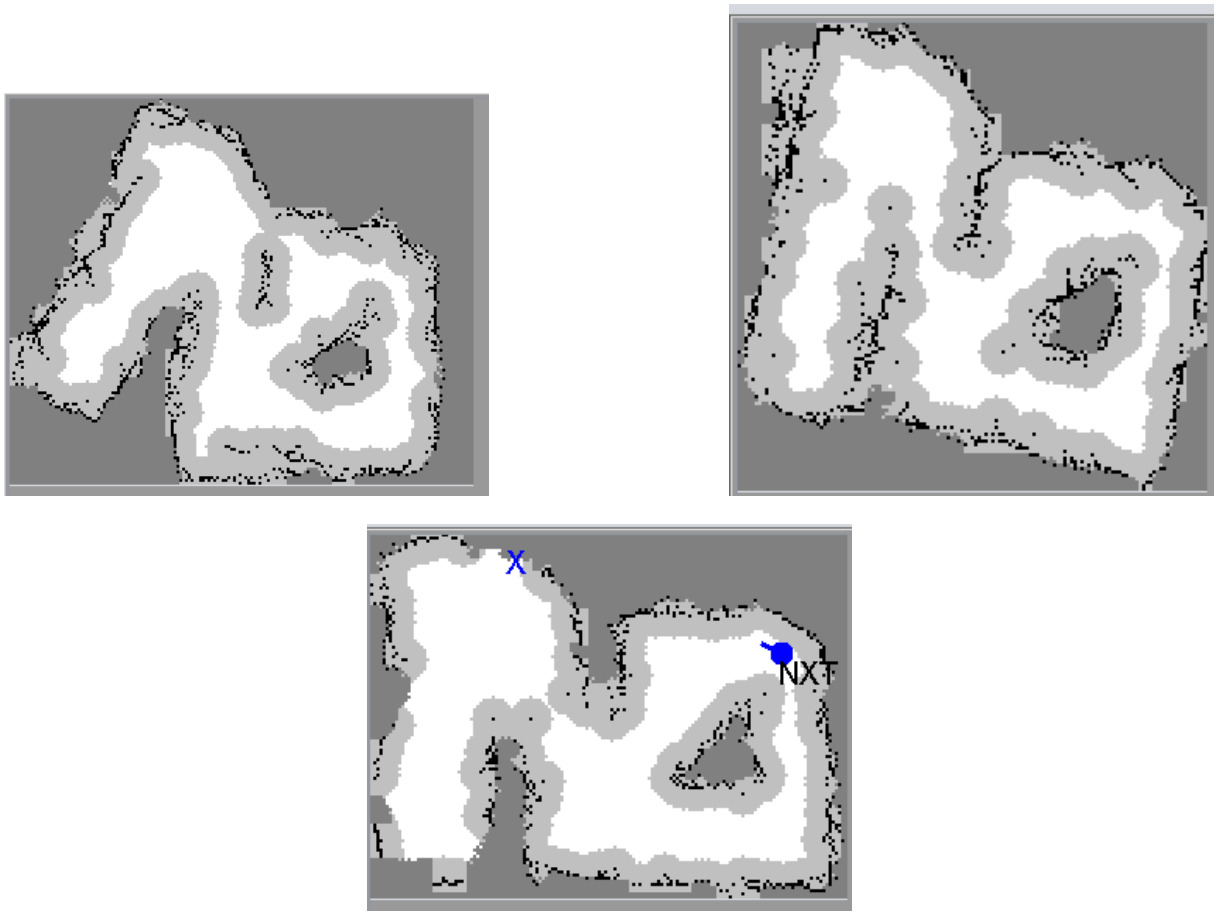
Figur 9.2: Testlabyrinten

Først ble labyrinten avgrenset slik at åpningene under og til venstre for trekanten ble stengt. Figur 9.3 viser kartene som ble produsert ved to forskjellige kjøringer i dette oppsettet.



Figur 9.3: Liten labyrint

Videre ble labyrinten utvidet slik at åpningen til venstre for trekanten ble frigjort, og istedet ble gangen til venstre blokkert nederst. Resultatene fra tre forsøk i denne vises i figur 9.4.



Figur 9.4: Middels labyrint

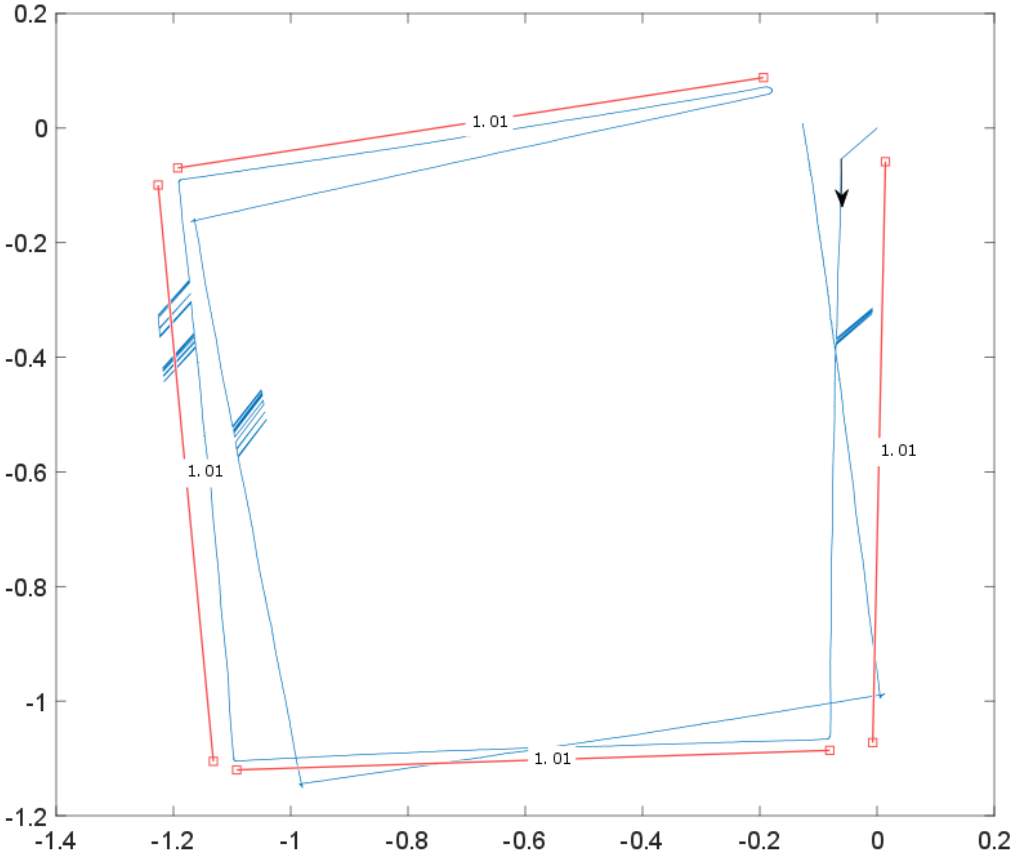
Til slutt ble det kjørt en test i hele labyrinten, med ingen blokkeringer. Se figur 9.5.

Figur 9.5: Test i hele labyrinten

Under denne testen er det åpenbart at roboten på et tidspunkt har kollidert med en vegg eller sklidd på underlaget, da det er en klar forskyvning i kartet.

I vedleggene ligger det videoer fra robotens kartlegging av denne labyrinten.

Figur 9.6 viser resultatet fra en kjøring der robotens virkelige posisjon har blitt logget av et motion-capture system. Roboten har her blitt bedt om å kjøre i en 1x1 m firkant først den ene veien deretter den andre. Pilen oppe i høyre hjørne viser startposisjonen. Et par av målingen var støyete og resulterte i en hakkete linje noen steder. De røde linjene måler lengden til firkantens kanter.



Figur 9.6: Kjøring i 1x1 m firkant

Kapittel 10

Diskusjon

10.1 IO-kretskort

Løsningen som er implementert med en egen mikrokontroller som innhenter IO-data for roboten fungerer veldig godt. Dette har løst problemet med hvordan alle de forskjellige enhetene kunne kobles til NXT, og kommunikasjonen med Bluetooth-dongelen er ikke lenger utsatt for kollisjoner mellom inn- og utgående data.

Hvordan data utveksles mellom NXT og IO-mikrokontrolleren er også veldig god og pålitelig. Måten all data fra de forskjellige taskene hos NXT, og fra IO-mikrokontrolleren flettes sammen på den halv-duplex kommunikasjonslinjen er solid.

Ett problem som kan oppstå er når kabelen mellom kretskortet og NXT ikke er plugget i, da hoper det seg opp meldinger som ikke blir sendt og send-buffere fyller seg opp og kræsjer systemet. Dette kan skje både dersom kabelen ikke er koblet i ved oppstart, og dersom den kobles ut underveis. Kabelen sitter godt fast, så dette skjer kun dersom man fysisk drar den ut. Problemet er derfor et lite et, men det burde vært implementert en handshaking mellom de to partene med jevne mellomrom, slik at det ikke gjøres forsøk på å sende data når IO-mikrokontrolleren ikke er tilgjengelig.

Dersom man ser stort på det, kan det også argumenteres for at bruken av både NXT og den eksterne mikrokontrolleren er unødvendig kompleks. I teorien kunne hele applikasjonen kjørt på denne mikrokontrolleren, og NXT er i så måte unødvendig. Roboten ville da blitt veldig lignende AVR-roboten. Det eneste NXT-spesifikke som brukes nå er servomotorene. En annen måte denne oppgaven derfor kunne vært vinklet, er å ha IO-mikrokontrolleren som

hovedenheten i systemet, og så sender denne servomotor-kommandoer til NXT. En slik løsning ville vært fullt mulig dersom IO-mikrokontrolleren hadde større minne, den som brukes nå har kun 1 kB flash-minne og kunne ikke inneholdt hele applikasjonen.

10.2 Kommunikasjon

Den nye kommunikasjonsløsningen er en forbring som gjør systemet mer fleksibelt og åpner for nye muligheter. Forbedringen består egentlig av to forskjellige elementer: den ene er nettverkslaget som implementerer rammer med adresser og gjør det mulig å sende meldinger til hvem som helst. Den andre er ARQ-protokollen i transportlaget som implementerer retransmisjon av tapt data.

Dette gir ikke bare fleksibilitet når det kommer til at data kan flyte fritt til hvilken som helst av enhetene på nettverket, men også inndeling i lag gjør at ny funksjonalitet er enklere å implementere. Det er for eksempel ikke noe problem å skrote hele transportlaget slik at applikasjonen benytter seg av nettverkslaget direkte. Dette er jo ikke noen god løsning da applikasjonen måtte tatt ansvar for flere aspekter ved sendingen, men oppsettet tillater der. Et mer fornuftig eksempel er at det kan lages nye protokoller i transportlaget som utnytter seg av nettverkslaget, uten at sistnevnte trenger å endres. Det har vært fokus på et klart skille mellom lagene slik at nettopp slike ting er mulig.

Nettverkslaget fungerer veldig godt og lite kan utsettes på hvordan dette fungerer. Det er dog noen utfordringer i ARQ-protokollen i transportlaget. Her er det kompleks funksjonalitet grunnet alle de mulige tilstandene som kan oppstå. Denne kompleksiteten åpner for uforutsette tilstander som ikke behandles. Det har aldri blitt observert feil det har vært mulig å spore tilbake til ARQ-protokollen, men det er mulig at ustabilitet som det hender forekommer har sin rot her.

Slik ARQ-forbindelser fungerer nå, kuttes forbindelsen ved tap av nok meldinger på rad, og roboten må restartes for å sette opp en ny tilkobling. Dette kunne blitt forbedret ved å utvide protokollen med funksjonalitet som prøver å gjenopprette kontakten. Det har dog aldri vært oppdaget tap av meldinger i en standardsituasjon, kun ved simulering av tapte pakker, så dette er et mindre problem.

Faktumet at det ikke har vært observert tapte pakker reiser også spørsmålet om retransmisjonsprotokollen i det hele tatt er nødvendig, og bare skaper unødvendig kompleksitet. Det

er greit å ha funksjonaliteten i bakhånd, men dersom det faktisk ikke skjer noen feil kan det vurderes å kun bruke den simple protokollen.

Bruken av et adressebasert nettverkslag er per nå ikke nødvendig, da applikasjonen ikke har behov for å sende meldinger mellom roboter, så det kan diskuteres hvor riktig det var å implementere dette. Bortsett fra mulighetene dette åpner for, som er nevnt flere ganger, er den eneste virkelige forbedringen at server-applikasjonen ikke lenger hver gang den skal sende noe først må sende en melding direkte til sentral-dongelen om destinasjonen til den påfølgende dataen.

10.3 Kartlegging

Bildene presentert i resultat-kapittelet viser tydelig at kartleggingen til roboten er langt fra optimal. Det er vanskelig å argumentere for at dette har blitt forbedret siden prosjektarbeidet til Lien [3]. Den eneste virkelig merkbare forskjellen er at roboten nå veldig sjelden kolliderer med veggene. Dette er et resultat av økningen fra to til fire avstandssensorer, som gjør at én av disse nesten alltid peker noenlunde rett fremover og kan advare om en forestående kollisjon.

Hvorfor ingen spesiell forbedring har skjedd kan forklares med at temaene denne oppgaven har tatt for seg ikke har vært relatert til videreutvikling og forbedring av applikasjonen, men har fokusert på underliggende deler i roboten.

Det eneste som virkelig hadde hatt mulighet til å gjøre en forskjell er at gyro- og kompassdata nå er tilgjengelige for bruk. Da de innledende forsøkene på å bruke denne dataen i posisjonsestimatoren resulterte i verre ytelse enn kun ved bruk av servoenkoderverdiene, som er veldig nøyaktige, ble det ikke brukt mer tid på å få feilsøke hva problemet var. Istedet brukes nå ikke denne dataen til noe. Hvis litt arbeid hadde blitt lagt ned i dette skulle det ikke være noe i veien for at NXT kommer på samme nivå som AVR.

Resultatet fra firkantkjøringen viser også at roboten ikke roterer den vinkelen den blir bedt om, men litt mindre. I løpet av kjøringen frem og tilbake har det blitt gjort åtte rotasjoner, alle litt for kort, og det resulterer i at avviket til slutt blir ganske stort. Resultatene viser også at distansen roboten kjører er litt lenger enn det kommandoen sier, ca 1 cm. Det positive å trekke ut fra testen er at roboten kjører helt rett uten å drifte den ene eller andre veien. Problemene med for lite rotasjon og for lang distanse skal det være mulig å forbedre ved å tilpasse

enda mer nøyaktig konstantene som forteller hvor mye ett enkoder-tikk utgjør i millimeter. Dersom gyroskopet og kompasset også blir fikset vil dette også hjelpe på feilrotasjonen.

Kapittel 11

Videre arbeid

11.1 Endre nettverkstopologien

Det skal være mulig å sette opp donglene til å forme et mesh-nettverk. Det burde gjøres undersøkelser for å finne ut om det er gjennomførbart.

Én mulighet er at donglene settes opp til å være periferi og sentral samtidig. Da kan en dongel være periferien til én og sentralen til en annen.

Eventuelt er det mulig å gå over til en helt annen nettverksteknologi enn Bluetooth. For eksempel ZigBee har innebygd støtte for mesh. Dette vil dog kreve veldig mye mer arbeid og tilpassing av de andre delene av systemet.

11.2 Posisjonsestimat

Ta i bruk dataen fra gyroskop og kompass i posisjonsestimatoren. Det har vært problemer med å få disse til å gjøre posisjonsestimatet bedre, så de tas ikke i bruk. Muligens må hele posisjonsestimatoren felles for robotene erstattes med en bedre én. Konstantene i defines.h som har innvirkning på hvordan enkoderverdiene fra hjulene gjøres om til millimeter bør også finstilles. De to som har noe å si for dette er distansen mellom hjulene og millimeter mer tikk.

11.3 IO-NXT kommunikasjon

Det burde implementeres logikk slik at NXT kan teste om IO-mikrokontrolleren er tilgjengelig, og hvis ikke stoppe å sende meldinger til den. Det ble gjort et enkelt forsøk på dette som ikke lykkes. Per nå kan programmet kræsje hvis IO ikke er koblet og NXT sender mange nok meldinger.

Tillegg A

Meldingstyper

Handshake (type 0)	
Felt	Bytes
Name length	1
Name	1-10
Width	2
Length	2
Tower offset x	1
Tower offset y	1
Axel offset	1
Sensor1 offset radius mm	1
Sensor2 offset radius mm	1
Sensor3 offset radius mm	1
Sensor4 offset radius mm	1
Sensor1 heading deg	2
Sensor2 heading deg	2
Sensor3 heading deg	2
Sensor4 heading deg	2
Deadline	2

Update (type 1)	
Felt	Bytes
X pos	2
Y pos	2
Heading	2
Tower angle	2
Sensor1	1
Sensor2	1
Sensor3	1
Sensor4	1

Order (type 2)	
Felt	Bytes
Orientation	2
Distance	2

Den neste listen oppsummerer ID til de forskjellige meldingstypene. Alle unntatt handshake, update og order, som har data med formatet beskrevet over, har ingen medfølgende data, og det er bare ID som sendes. For handshake, update, og order kommer ID først, deretter data i den rekkefølgen de står i listene over. For felter på mer enn én byte kommer bytene i little endian rekkefølge: den lavest byten sendes først. For eksempel dersom lengden på roboten er 100 mm, 0x0064 hexadesimalt, sendes dette som 0x64 0x00.

Ping og ping response er definert, men brukes ikke i applikasjonen.

Type ID	
Handshake	0
Order	1
Update	2
Idle	3
Pause	4
Unpause	5
Confirm	6
Finish	7
Ping	8
Ping response	9
Debug	10

Eksempel

En order melding som ber roboten rotere 270 grader og kjøre 200 mm fremover, vil se slik ut: 0x01 0x0E 0x01 0xC8 0x00.

Felt	Desimalt	Hex
Type	1	0x01
Vinkel	270	0x010E
Avstand	200	0x00C8

En Idle melding derimot, vil kun se slik ut: 0x03

Tillegg B

Meldingseksempel

En order melding (type 1) som ber roboten rotere 270 (0x0E01) grader og kjøre 200 (0x00C8) mm fremover, vil se ut som bytene i den første raden i tabellen under, når man bruker little endian rekkefølge på felt med mer enn én byte.

Raden i midten viser hvordan meldingen ser ut når den forlater transport-laget. Segment-typen for data er 0, og her er sekvensnummeret antatt å være 34 (0x22).

Den nederste raden viser meldingen etter å ha blitt behandlet av nettverkslaget, rett før den blir COBS-enkodet og sendt. Adressen til avsender er satt til 0 og mottakeraddressen er 1.

Fargene viser hvilke byte som er lagt til av de forskjellige lagene.

Avs.	Mot.	Prot.	Seg.-type	Sekv.nr	Len.	Type	Vinkel	Avstand	CRC
						0x01	0x0E 0x01	0xC8 0x00	
			0x00	0x22	0x05	0x01	0x0E 0x01	0xC8 0x00	
0x00	0x01	0x00	0x00	0x22	0x05	0x01	0x0E 0x01	0xC8 0x00	0x01

Til slutt blir bytene fra nettverkslaget (nederste rad) COBS enkodet for å ta bort nuller i meldingen, og lagt til 0x00 på slutten. Bytene som faktisk sendes over nettverket vil da være disse:

0x01 0x02 0x01 0x01 0x07 0x22 0x05 0x01 0x0E 0x01 0xC8 0x02 0x01 0x00.

Tillegg C

Beskrivelse av vedlegg

C.1 1. Bilder og video

Bilder og video av robotens kartlegging.

C.2 2. Kildekode

Kildekode for serverapplikasjon, NXT og IO-mikrokontroller.

C.3 3. Kretskortdesign

EAGGLE-designfiler for kretskortet.

C.4 4. Bruksanvisninger

Instruksjoner og guider for ting som kan være greit å vite. Hvordan Bluetooth-donglene kan programmeres, introduksjon til NXT-roboten mm.

C.5 5. Datablad og manualer

Datablad for komponenter og dokumentasjon for forskjellig utstyr.

C.6 6. Tidligere rapporter

Alle tidligere rapporter og vedlegg

Bibliografi

[1] LEGO Group. LEGO MINDSTORMS NXT Hardware Developer Kit.

[2] Maxim Integrated. Understanding and using Cyclic Redundancy Checks with Maxim iButton Products. http://www.sal.wisc.edu/pfis/docs/rss-nir/archive/public/Product%20Manuals/maxim-ic/AN27.pdf. Lastet ned: 2017-05-24.

[3] Kristian Lien. Fjernstyring av legorobot, 2016.

[4] Stuart Cheshire og Mary Baker. Consistent Overhead Byte Stuffing. http://www.stuartcheshire.org/papers/COBSforToN.pdf. Lastet ned: 2017-05-31.