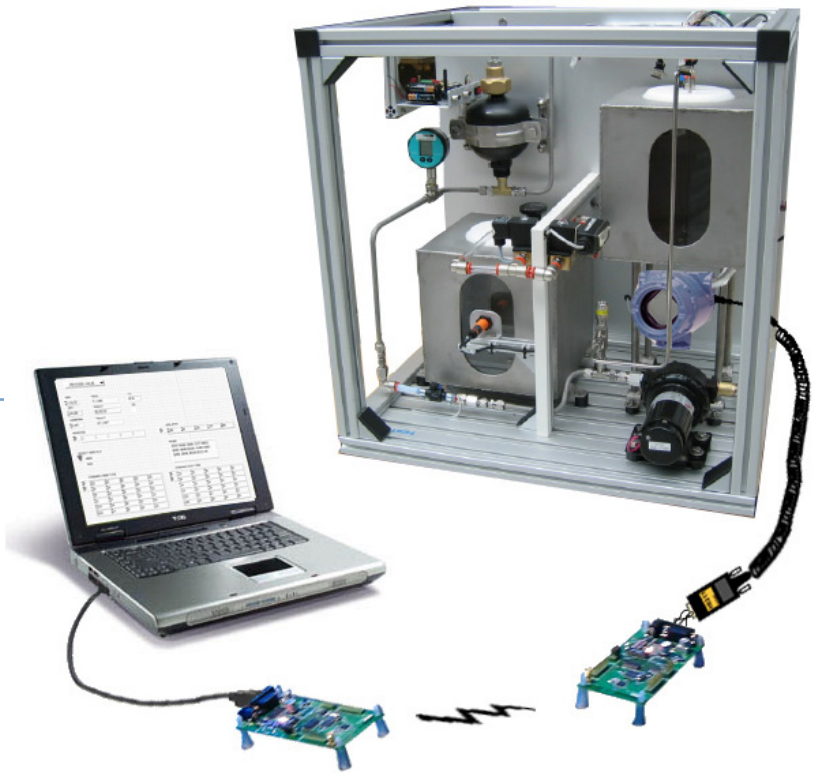
Prosjektoppgave

Nils Petter Eftedal

Trådløs overføring av HART via ZigBee

Trondheim desember 2005

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultetet for informasjonsteknologi, matematikk og elektroteknikk
Institutt for teknisk kybernetikk



NTNU
Det skapende universitet

Nils Petter Eftedal

Trådløs overføring av HART via ZigBee

Prosjektoppgave

Trondheim, desember 2005

Norges teknisk-naturvitenskapelige universitet
Fakultetet for informasjonsteknologi, matematikk og elektroteknikk
Institutt for teknisk kybernetikk

Hovedveileder: Tor Onshus

NTNU
Det skapende universitet

NTNU
Norges teknisk-naturvitenskapelige universitet

Fakultet for informasjonsteknologi, matematikk og elektroteknikk
Institutt for teknisk kybernetikk

PROSJEKTOPPGAVE

Kandidatens navn: Nils Petter Eftedal

Fag: Teknisk Kybernetikk

Oppgavens tittel (norsk): Trådløs overføring av HART data via ZigBee

Oppgavens tittel (engelsk): Wireless transmission of HART data via ZigBee

Oppgavens tekst:

ZigBee standardens inntog på det industrielle markedet har gjordt det mulig å utnytte eksisterende teknologier på nye områder. Ved hjelp av ZigBee blir det for eksempel mulig å hente inn data fra sensorer som sitter på roterende enheter samt foreta overføringer på industrielle områder hvor kabling før har vært en utfordring. Trådløs overføring vil også være økonomisk besparende med tanke på kablingskostnadene man slipper.

Denne oppgaven vil ta for seg videreføring av data fra et HART nettverk ved hjelp av ZigBee.

- Sette seg inn i HART og ZigBee overføringer
- Lage et grensesnitt mellom HART og ZigBee
- Hente ut og overføre HART-data mellom to ZigBee-noder

Oppgaven gitt: 22. august 2005

Besvarelsen leveres innen: 20. desember 2005

Utført ved Institutt for teknisk kybernetikk

Veileder: Knut-Olav Fjell, Niels Aakvaag

Trondheim, 31. august 2005

Tor Onshus
Faglærer

Forord

Hensikten med prosjektet har vært å foreta en fysisk implementering som muliggjør trådløs kommunikasjon mellom *master* og slave i et HART-nettverk. Rapporten vil dokumentere arbeidet som er blitt gjort og gi nødvendig kunnskap for å rekonstruere eller videreby gge på de resultatene som er fremkommet. Arbeidet er utført delvis ved NTNU og delvis ved Statoils forskningssenter på Rotvoll.

Det er mange personer som har vært med på å gjøre dette prosjektet både kunnskapsrikt og interessant. Jeg vil gjerne takke Niels Aakvaag og Knut-Olav Fjell som har vært med på å gjøre denne oppgaven mulig å gjennomføre. Uten utstyr og programvare ville ikke prosjektet vært på langt nær så innholdsrikt. En stor takk går også til lærlingene Linda Vrangen og Anja Engtrø for oppgradering av ZigBee-riggen. Anja har også hjulpet til med utstyr underveis og har aldri vært mer enn noen tastetrykk unna når noe har manglet. Gisle H. Bedin skal heller ikke glemmes. Han har blant annet hatt oppdraget med valg av innkjøpt utstyr og enheter.

Ellers vil jeg også takke Magne Sætre for hjelp med driveren i LabVIEW. Uten hans innspill og kompetanse ville prosjektet fått en mye slettere startkurve. Til sist, men absolutt ikke minst vil jeg rette en stor takk til professor Tor Onshus for veiledning og støtte gjennom hele prosjektet.

Nils Petter Eftedal

Trondheim, desember 2005.

Sammendrag og konklusjon

Sammendrag

Nye trådløse protokoller med interessante egenskaper for instrumenteringssystemer er for tiden i sterk medvind. Den nye LR-WPAN standarden IEEE 802.15.4 har ført til en god plattform for mer avanserte nettverksprotokoller med lavt energiforbruk og lave kostnader. I denne rapporten vil ZigBee bli benyttet og implementert for kommunikasjon mot HART. Trådløse overføringer vil kunne tilføre denne feltbuss-standarden reduserte kostnader i form av redusert kabling og mulighet for innhenting av data fra sensorer plassert på bevegelige og vanskelig kablede gjenstander.

Arbeidet med oppgaven er utført i samarbeid med Statoil forskningssenter og ABB Corporate Research Center og er utført på forskningssenteret på Rotvoll i Trondheim. Deres bidrag har vært innkjøp av utstyr og enheter som har gjort den fysiske delen av prosjektet mulig å gjennomføre. Blant annet har en demonstrasjonsrigg blitt oppgradert med en trykksensor kompatibel for HART slik at ZigBee-noder kunne kobles mot denne og overføre meldinger til datamaskin.

Oppgaven vil som sagt fokusere på implementering. Dette ble foretatt i små steg frem til det endelige resultatet sto ferdig. Det er blitt sett på grensesnitt mot HART-sensoren og mulige måter å implementere dette. Under arbeidet med prosjektet har det imidlertid blitt benyttet et standard HART-modem. Første skritt i implementeringen har vært å få dette opp å gå og få kommunikasjon til å fungere mot en *master* i form av en datamaskin. De ulike kommunikasjonsprogrammene som har blitt testet er blitt beskrevet og den endelige valgte driveren, skrevet for labVIEW, er blitt grundig dokumentert. Problemstillinger som meldingsfeil fra sensor og dårlig timing av modemkontroll er spesielt vektlagt.

Det neste implementeringstrinnet som ble sett på var kommunikasjon mellom to ZigBee-noder. Den ene av disse ble implementert som *gateway* - for innsamling av data til PC, mens den andre ble implementert som et endepunkt for kommunikasjon mot sensor. Utvikling av programmene ble utført med ProfileBuilder som genererer kildekode for videre påbygning.

Siste steg i prosjektet bestod av oppkobling og optimalisering av det fullstendige systemet. ZigBee-kommunikasjonen ble finjustert ved å sette timingparametere så nær de kritiske grensene som mulig. Minimalt *overhead* som følge av ZigBee-nodene kan da beregnes til å være i overkant av 20ms uten å ta hensyn til den trådløse overføringen. Dersom det er ønskelig med determinisme ved bruk av

beacons må ytterligere 15,36ms legges til, men dette trengs kun når vi har fler noder i systemet.

Konklusjon

I et system med harde sanntidskrav ville kanskje de ekstra millisekundene ZigBee skaper være et problem, men i et system basert på HART hvor kommunikasjon foregår med en hastighet på 1200 baud vil dette trolig ha lite å si. Den virkende løsningen som ble resultatet av dette prosjektet kan altså benyttes til instrumentering avhengig av kravene som stilles. Ved store mengder noder vil vi forøvrig få samme problemer som med ZigBee generelt; determinisme kan ikke garanteres for mer enn 7 noder gjennom garanterte tidsluker og *beacon*sending.

Videre Arbeid

Det er flere oppgaver som må utføres før dette prosjektet kan videreføres til et virkelig instrumenteringssystem. For å redusere pris og fysisk størrelse bør det lages kretskort hvor kun nødvendige komponenter settes sammen og hvor også HART-modem er tatt med i designet. Driveren på datamaskinsiden kan ytterligere forbedres, blant annet er seriekontroll og timing av dette nå overflødig da ZigBee-nodene har overtatt denne oppgaven. Det er også mulig med ytterligere reduksjon av *overhead* i nodene ved å bygge opp egne funksjoner for innhenting av seriedata (med uart) istedenfor å benytte den definerte seriemodulen medfølgende *stacken*. Det vil gjennom denne rapporten bli gitt kunnskap og grunnlag for å utføre alle forbedringene nevnt ovenfor.

Dersom det er ønskelig å innhente data fra HART-nettverk koblet i multidrop bør data mottatt fra ulike sensorer settes sammen til større meldinger før overføringer. Dette gjelder spesielt om det er flere noder i ZigBee-nettverket. Ved å implementere dette vil sjansene for kollisjoner reduseres og vi vil få mindre *overhead* i systemet.

Innhold

Figurer

Tabeller

Kapittel 1

Innledning

1.1 Bakgrunn

Trådløs kommunikasjon har med årene blitt både sikrere, rimeligere og mindre energikrevende. Alle disse egenskapene er viktige når trådløse sensornettverk skal vurderes til bruk i industrielle sammenhenger. Klare fordeler som reduserte kablingskostnader, fleksibilitet og mulighet for innhenting av data fra roterende og vanskelig kablede enheter gir dette mediet en klar fordel sammenlignet med tradisjonell kabling. Det er derfor interessant å se på egenskapene til nyutviklede trådløse teknologier for å se om de har noe å tilføre på områder som før har vært dominert av trådbundet kommunikasjon. I samarbeid med Statoil og ABB har en slik oppgave blitt utformet.

ZigBee er en relativt ny trådløs standard med mange interessante egenskaper for industriell kommunikasjon. Dette prosjektet vil benytte denne standarden og implementere den for kommunikasjon mot HART. Trådløse overføringer vil ha mye å tilføre denne feltbuss-standarden i form av egenskapene nevnt i forrige avsnitt.

1.2 Problemstilling og endelig mål

I hovedsak vil dette prosjektet dreie seg om å implementere et testoppsett med HART hvor innsamling av data foregår via ZigBee. Ved Statoils forskningssenteret på Rotvoll er en testrigg blitt oppdatert med en trykksensor kompatibel for HART. En ZigBee-node vil bli koblet til riggens sensorsløyfe gjennom et hittil udefinert grensesnitt, kommuniserende med en annen ZigBee-node koblet mot en datamaskin. Det endelige målet er at kommunikasjonen skal foregå som om den var trådbundet. Ut i fra dette kan slutninger trekkes om utvidede muligheter for denne *gatewayen*.

1.3 Disposisjon

Rapporten er inndelt i tre hovedbolker etterfulgt av vedlegg og bibliografi. Den første bolken utgjør forstudiet til prosjektet, her vil oppbygning og virkemåte

av HART og ZigBee bli studert nærmere. For både ZigBee og HART vil mye av teorien bli satt mer i sammenheng og bli ytterligere utdypet når vi går videre til neste bolk. Fokus for denne er selve implementeringsdelen av prosjektet og det er også her hovedtyngden av arbeid har blitt nedlagt. I den siste bolken vil de endelige resultatene bli fremlagt og diskutert.

Del I

Forstudie

Kapittel 2

Bakgrunnsteori

Dette kapittelet er tatt med for å gi en forklaring på grunnleggende begreper som er nødvendige for å forstå de kommende kapitlene. ISO/OSI standarden er tatt med som en modell vi bør ha kjennskap til da denne er grunnleggende i de fleste protokollers oppbygning. En innledning og forklaring av sammenhengen mellom ZigBee og IEEE 802.15.4 vil deretter bli gitt. Det anbefales sterkt å lese dette før kapitlene for de to blir lest.

2.1 OSI modellen

For at enheter i et nettverk skal kunne kommunisere er de nødt til å "snakke samme språk", noe som vil si at de må tolke meldinger på samme måte og kunne benytte samme transportmedium for overføringer.

Det første skrittet med å standardisere ulike kommunikasjonsnettverk ble gjort av International Standards Organization (ISO) i 1978. De presenterte da den lagdelte Open Systems Interconnect (OSI) modellen som beskriver hvordan protokoller kan deles opp i syv lag. Hovedtanken er at hvert lag yter tjenester til laget ovenfor uten at denne trenger å vite noe om hvordan disse er implementert. Med dette oppnår vi blant annet at et lag kan skiftes ut med et annet, så lenge grensesnitt og tjenester opp/ned i referansemodellen holdes uforandret. En annen velkommen bivirkning er at selve implementeringsarbeidet med en protokoll vil kunne deles opp i syv naturlige deloppgaver, hvorav arbeidet med disse kan gjøres parallelt. Det vil altså oppnås hurtigere utvikling med et mer forståelig og strukturert sluttresultat. Modellens oppbygning er vist i *figur 2.1* hvor det nederste laget er lag 1 og økende oppover i pyramiden. Nedenfor vil det bli gitt kort forklaring av de ulike lagene.

Applikasjonslag: Inneholder applikasjoner og definerer grensesnittet mot bruker.

Presentasjonslag: Gir et felles format for datapresentasjon. Blant annet blir kompresjon og kryptering foretatt i dette laget.

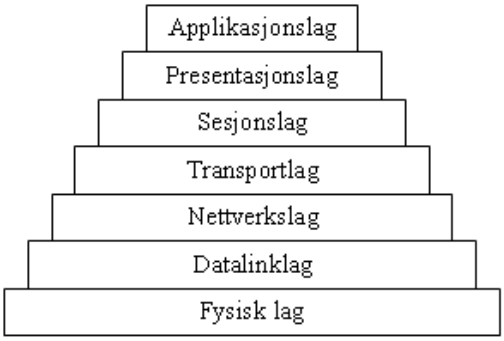
Sesjonslag: Etablerer, kontrollerer og avslutter dialog mellom sender og mottaker. Inneholder også synkronisering og ressurstildeling.

Transportlag: Sørger for at informasjon blir sendt og mottatt riktig[1]. Deler blant annet store meldinger i så små biter at nettverkslaget kan håndtere dem. Dette gjelder også motsatt vei når meldinger fra nettverkslaget skal settes sammen igjen.

Nettverkslaget: Angir rammebetingelsene for trafikken i nettverket. Routing av pakker og lignende.

Datalinklaget: Pakker data i rammer og etablerer kommunikasjon. Dette innebærer oppdeling av melding slik at fysisk lag klarer å overføre dem og motsatt vei ved å behandle og sette sammen den rå bitstrømmen mottatt fra fysisk lag. Etablering av kommunikasjon innebærer metoder for nettaksess for eksempel CSMA/CD. Laget er oppdelt i to underlag; *Logical Link Control* (LLC) og *Media Access Control* (MAC) som tar seg av oppgavene i laget i samme rekkefølge som de er anngitt ovenfor.

Fysisk lag: Maskinvare som overfører data.



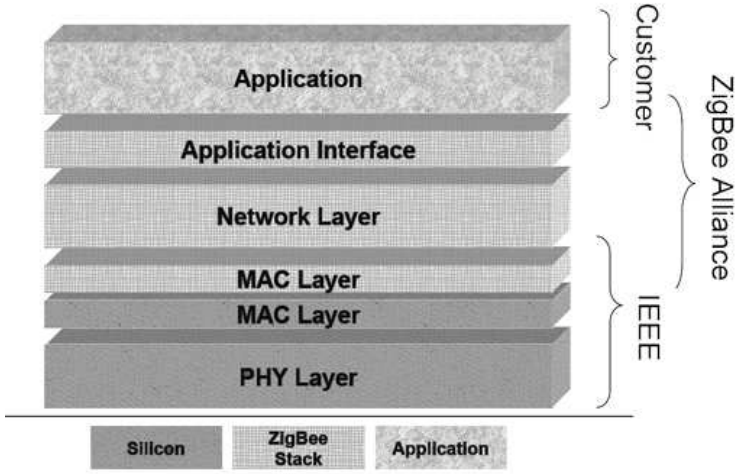
Figur 2.1: OSI-modellens oppbygning.

2.2 ZigBee og IEEE 802.15.4

ZigBee er utviklet av ZigBee Alliance som et nytt trådløst alternativ til bruk i lavstrømsapplikasjoner med lav datarate. ZigBee Alliance arbeid bygger på 802.15.4 standarden utviklet av IEEE (*Institute of Electrical & Electronic Engineers*). Denne standarden definerer PHY og MAC-lagene i OSI modellen og utgjør fundamentet for et LR-WPAN (*Low-Rate Wireless Personal Area Network*). ZigBee-*stackens*[2] arkitektur er vist i *figur 2.2*. Den definerer kun de lagene som er nødvendig for å oppnå ønsket markedsfunksjonalitet. Presentasjon-, sesjon- og transportlag (lag 6,5 og 4) er utelatt.

[1] Dette kalles Quality of Service (QoS).

[2] En *stack* er definisjonen på selve implementeringen av lagene i en protokoll.



Figur 2.2: ZigBee *stack*, modifisert fra [4].

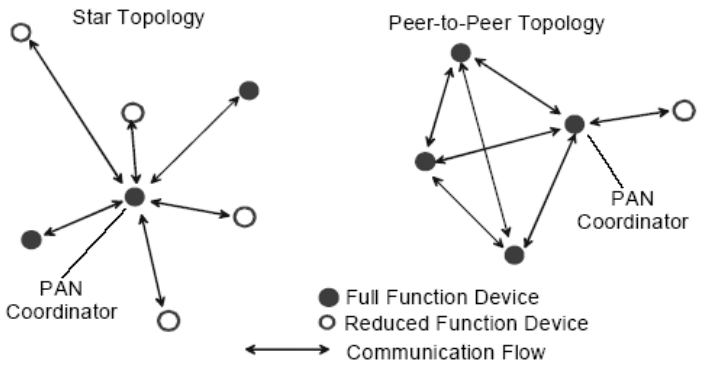
Kapittel 3

IEEE 802.15.4

Motivasjonen for å utvikle IEEE 802.15.4 standarden var å få et LR-WPAN med lavt energiforbruk og ekstremt lave kostnader. Den er beregnet på kommunikasjon over korte avstander med en POS (*Personal Operating Space*) på rundt 10 meter. Andre egenskaper som knyttes til LR-WPAN er lave datarater[1] samt støtte for både stjerne og *peer to peer* topologier.

3.1 Komponenter og topologier

For å redusere energiforbruk har et LR-WPAN to forskjellige enhetstyper. En enhet kan enten være av typen *full functional device* (FFD) eller *reduced functional device* (RFD). En FFD kan operere i tre forskjellige modi, som PAN- (*Personal Area Network*) koordinator, koordinator, eller som en vanlig enhet. Som det ligger i navnet inneholder en RFD minimalt med funksjonalitet og kan kun benyttes til å kommunisere med en FFD. I *figur 3.1* er enhetstypene vist slik de kan kobles i de tilgjengelige topologiene.



Figur 3.1: Stjerne og *peer to peer* topologi hentet fra [5].

[1] Dataratene avhenger av frekvensbåndet det sendes i og er oppsummert i *tabell 3.2* sammen med andre egenskaper for båndene.

PAN-koordinatorens oppgave er å starte opp nettverket og koordinere ulike oppgaver. PAN-koordinatoren kan sende *beacons*, noe som vil forklares senere. Koordinatoren har egenskaper som en PAN-koordinator, men kan ikke starte et nettverk. Den kan benyttes til å videreruter meldinger og kan konvertere til PAN-koordinator om denne skulle gå ned. En vanlig enhet har ingen routingsfunksjonalitet og kan kun benyttes i endepunkter, kommuniserende med en PAN-koordinator eller koordinator.

3.2 Det fysiske laget (PHY)

Det fysiske laget inneholder en RF-sender sammen med lavnivå kontrollmekanismer for denne. Tre alternative fysiske lag er definert. Egenskapene for hver av disse avhenger av hvilket frekvensbånd som benyttes. Et kort sammendrag av disse er gitt i *tabell 3.1* etterfulgt av en oppsummering av lagets viktigste oppgaver, hentet fra [5].

Frekvensbånd	2450 MHZ	915 MHz	868 MHz
Datarate	250 kb/s	40 kb/s	20 kb/s
Antall kanaler	16	10	1
Kanalseparasjon	5 MHz	2 MHz	N/A
Lisensfritt	Globalt	USA	Europa
Modulasjonsteknikk	Q-QPSK	BPSK	
Rekkevidde	10-100m	20-100m	

Tabell 3.1: Egenskaper for ulike frekvensbånd, rettet opp fra [15].

Av/på: Aktivering og deaktivering av senderen.

Frekvensvalg: Valg av kanal.

Dataoverføringer: Sending og mottak av data.

***Energy Detection* (ED):** Estimering av mottatt signalstyrke. Dette gjelder også signaler som ikke stammer fra en 802.15.4 sender. En måte å sikre oss mot forstyrrelser fra andre signalkilder.

***Link Quality Indication* (LQI):** En LQI-måling foregår hver gang en pakke mottas. Det vil si å sjekke linkens styrke eller kvalitet. Dette kan foregå ved bruk av ED, en signal til støy ratio, eller en kombinasjon av disse. LQI er for eksempel viktig når optimal vei til destinasjon skal velges (*mesh* topologi).

Clear Channel Assessment (CCA): Vil si å lytte på en kanal for å se om den er ledig. Er en viktig del av CSMA-CA algoritmen 803.15.4 standarden benytter seg av ved sending og mottak.

3.3 MAC sublaget

MAC sublaget kan benytte seg av all funksjonaliteten det underliggende laget har å tilby. Den spesifiserer meldingssendinger og funksjonalitet gjennom grensesnittet mot det fysiske laget. Som for det fysiske laget står de viktigste egenskapene listet opp i [5].

***Beacons*:** Dersom enheten er en koordinator kan den sende ut *beacons*. I korte trekk innebærer dette å sende ut et signal som tillater kommunikasjon i et bestemt tidsrom. I den inaktive perioden mellom *beacons* (om noen) kan store strømbesparelser gjøres ved å sette nodene i *sleep modus*. Utsending av *beacons* innebærer at MAC laget også må kunne ta seg av synkronisasjon mot disse.

***Guaranteed Time Slots* (GTS):** Dersom en *beacon*-ramme er satt opp med en konkurransefri (CFP) periode kan det sikres kommunikasjon for en applikasjon ved å sette opp en garantert tidsluke (GTS). MAC laget har ansvaret for håndtering og vedlikehold av GTS mekanismen.

Forbindelser og sikkerhet: Kunne opprette og fjerne PAN-forbindelser samt gi støtte for enhetssikkerhet.

Kanalaksess: Utføre CSMA-CA for kanalaksess.

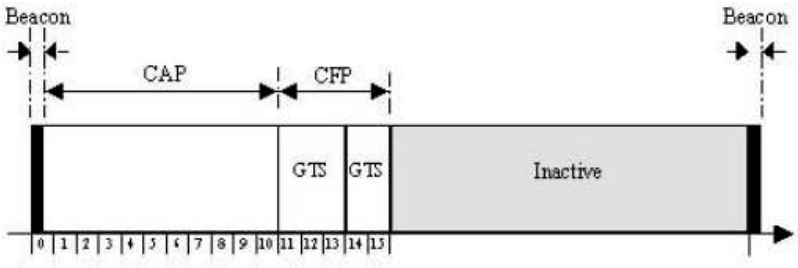
Pålitelig link: Opprette en pålitelig link mellom to enheters MAC lag. For å oppnå dette benyttes blant annet frame acknowledgement og retransmisjon, feildeteksjon ved hjelp av 16-bits CRC, samt CSMA-CA.

3.4 Kommunikasjon

IEEE 802.15.4 definerer to former for kommunikasjon, enten med eller uten *beacons*. Som nevnt i forrige seksjon fungerer becons ved at det sendes ut en regelmessige superramme hvor starten av hver ramme kjennetegnes av et *beacon*. Superrammen består av en aktiv del på 16 tidsluker, samt en valgfri ikke aktiv del. Den aktive delen kan bestå av to perioder; *contention acsess period* (CCA) hvor enheter kan konkurrere om sending, eller *contention free period* (CFP) hvor enheter kan få garanterte tidsluker (GTS). CFP kan romme opptil syv garanterte tidsluker. I *figur 3.2* er det vist et eksempel på en slik superramme.

I CAP benyttes en slottet versjon av CSMA-CA. Hver gang en enhet ønsker å sende data i denne perioden må den finne slutten på neste *back off*-luke, for deretter å vente et tilfeldig antall *back off* luker. Enheten må deretter lytte på kanalen for å vurdere om den er ledig for aksess. Er den ikke det må den vente et nytt antall *back off*-luker før den forsøker på nytt. Illustrasjoner og forklaringer er gitt i [5].

Velges alternativet uten *beacons* benyttes standard CSMA-CA uten tidsluker. Ønsker en enhet å sende data må den vente et tilfeldig tidsrom for deretter å



Figur 3.2: Eksempel på superrammestruktur hentet fra [5].

lytte på kanalen for å se om den er ledig. Er den ikke det må den vente et nytt tilfeldig tidsrom før den forsøker på nytt.

Kapittel 4

ZigBee

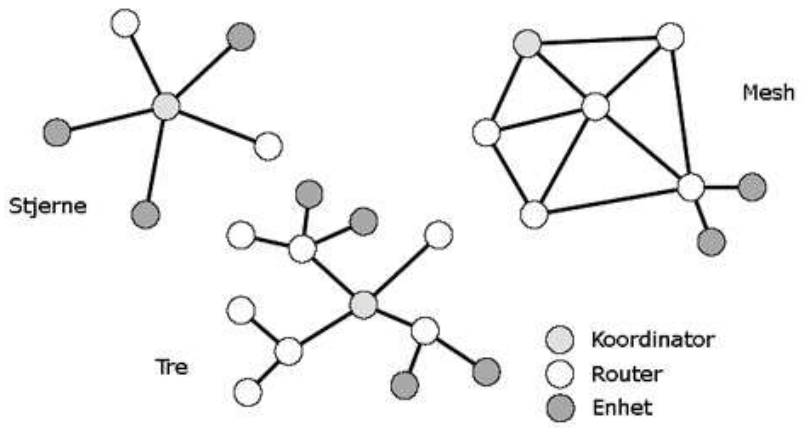
Som det ble sett på i forrige kapittel definerer IEEE 802.15.4 standarden et godt grunnlag for videre utvikling. ZigBee-alliansen har benyttet standarden som et fundament og bygget på med lag for nettverk, sikkerhet og rammeverk for applikasjonslaget. I de neste avsnittene vil det bli gitt en kort forklaring på funksjonaliteten i disse forskjellige *stack*-lagene. For mer detaljerte beskrivelser henvises det til *ZigBee Specification*, [1].

4.1 Nettverkslaget

Nettverkslaget inneholder funksjonalitet knyttet til kommunikasjon og dataoverføringer. Dette innebefatter oppgaver som adressering, routing og nabodeteksjon samt konfigurasjon avhengig av enhetstype. Eksempler på sistnevnte er tilkobling, frakobling og start av nettverk. Det er bare en PAN-koordinator som kan starte et nettverk. I ZigBee-terminologi vil denne hete ZigBee-koordinator, mens en vanlig koordinator nå vil hete en ZigBee-router. Vanlige enheter vil kalles ZigBee-enheter. Gjennom resten av oppgaven vil koordinator, router og enhet benyttes som betegnelse for disse.

Topologier

ZigBee støtter tre ulike nettverkstopologier; stjerne, mesh og trestruktur. Disse er vist i *figur 4.1*. Hver topologi har både fordeler og ulemper. Stjernenettverk gir langt batteriliv da alle noder unntatt kordinator kan være RFD-enheter. I et *beacon-enablet* nettverk vil disse kunne gå i strømsparingsmodus mellom hver sending. Mesh tilbyr økt sikkerhet da pakker kan routes alternative ruter (AODV-algoritme benyttes). Bakdelen er at det forutsetter at enheter til enhver tid må være aktive. *Beacons* kan altså ikke benyttes i denne topologien. Trestruktur utnytter noen av fordelene til både stjerne og mesh. Kommunikasjon kan foregå over lengre avstander ved å benytte routere. I motsetning til mesh kan også *beacons* benyttes.



Figur 4.1: Topologier tilbudt av ZigBee nettverkslag.

Adressering

Meldinger i et ZigBee-nettverk adresseres i en av tre ulike former; direkte, indirekte eller ved kringkastning. Avsnittene under vil forklare begrepene i den rekkefølgen de er nevnt.

Ved direkte adressering benyttes enten kort adresse på 16 bit eller den 64 bits lange MAC-adressen[1] som er unik for hver enhet. Kort adresse viser til den nettverksspesifikke adressen mottatt fra koordinator ved oppstart, mens MAC-adressen finnes i *hardware*. Med denne adresseringsformen kan enheter kommunisere direkte uten at *binding* eller *device discovery* trenger å foregå før sendinger kan starte. Neste avsnitt vil forklare disse uttrykkene nærmere.

Indirekte adressering innebærer at meldingen slipper å inneholde mottakeradresse da applikasjonen er linket til en eller flere mottagere gjennom en *bindingtabell*. Tabellen opprettholdes av koordinator og forteller hvilke enheter som tilbyr ulike tjenester i nettverket. Før kommunikasjon kan foregå må enheten sende en *binding request* til koordinatoren for å registrere tjenester og linkes til eventuelle enheter som tilbyr tjenestene vi er ute etter. *Device Discovery* er et lignende alternativ hvor søk etter tjenester vil bli foretatt.

Kringkasting oppnås ved å sette destinasjonsadressen til 0xFFFF og sendemodus til kringkastning (*broadcast*). En melding sendes da til alle enheter i nettverket. Adresseringsformen bør forøvrig benyttes med varsomhet og begrenses til bruk i ikke-kritiske nettverk på grunn av den ekstra båndbredden den vil oppta.

4.2 Applikasjonslaget

Applikasjonslaget er oppdelt i tre deler; *Application support sub-layer* (APS), *ZigBee Device Object* (ZDO) og de brukerdefinerte applikasjonsobjektene. Sist-

[1] MAC-adressen blir ofte også kalt IEEE-adresse.

nevnte er innbundet av *Application Framework* (AF) som danner et rammeverk både for egendefinerte og standardiserte applikasjoner.

***Application support sub-layer*:** utgjør et grensesnitt mellom nettverkslaget og applikasjonslaget ved å yte tjenester både for ZDO og brukerapplikasjoner. Blant oppgavene den har ansvar for finner vi kryptering og levering av pakker til rett applikasjon via AF. Leveringen baserer seg på pakkens *destination endpoint*. APS tar seg også av vedlikehold av bindingtabeller (koordinator). *Bindings* muliggjør sammenknytting av enheter basert på applikasjonene deres. Ved oppslag i bindingtabellen kan meldinger *routes* til destinasjon uten å benytte adresser. Altså indirekte adressering.

***Application Framework*:** Utvikleren av applikasjoner ser kun grensesnittet mot AF. Her velges det meldingsformat og grensesnitt for meldingssendinger. Noen flere nøkkelord vil nevnes i neste seksjon.

***ZigBee Device Object*:** Her defineres det blant annet hva slags enhetstype vi har; koordinator, router eller vanlig enhet. Sikker link mellom enheter vil bli initiert/etablert avhengig av gitte enhet. ZDO kjører som en vanlig applikasjon registrert mot *endpoint* 0 og tilbyr funksjoner for binding, sikkerhet, nodeadministrasjon og *discovery*. Sistnevnte går ut på søke for å lokalisere andre enheter og tjenester på nettverket. Meldinger til enpoint 0 følger profilen "*ZigBee Device profile*".

4.3 Applikasjonsdesign

For at kommunikasjon mellom ulike noder i et nettverk skal kunne skje må de forskjellige enhetene operere med samme meldingsformat og konfigurasjon. I ZigBee benyttes det profiler for å sørge for at kommuniserende applikasjoner ”snakker samme språk”. Applikasjoner på separate enheter kan med dette sende meldinger, motta data og prosessere kommandoer som en kommuniserende og distribuert applikasjon. I [1] er det gitt et eksempel på dette i form av en varmeprofil. Her kommuniserer en termostat på en node med en glødetråd på en annen node. I ZigBee-terminologi vil glødetråden og termostaten være to ulike enheter (*user devices*). For hver enhet kan det legges til et nødvendig antall clustere med så mange underliggende attributter (variabler) som ønskelig. Hver cluster er assosiert med data som strømmer inn eller ut av en enhet. Alle med sin egen unike cluster identifier innenfor den spesifikke profilen. I eksempelet med varmeprofilen er det temperaturinformasjon som skal oversendes. Hver enhet vil da typisk ha en temperaturcluster hvor cluster identifier er definert som ut i den ene og inn i den andre. Ved *binding* er det slike clustere som vil matches og bindes sammen. Kommunikasjon vil da kunne foregå uten adresser (indirekte adressering).

ZigBee terminologi

Flere ulike definisjoner som benyttes innenfor en profil ble nevnt innledningsvis. I denne seksjonen vil denne terminologien bli ytterligere utdypet.

På det laveste nivået i en profil finner vi attributtene. Dette er datainstanser som representerer en fysisk størrelse eller tilstand. Eksempler på dette kan være sensorverdier, lampe av/på osv. Disse verdiene blir sendt til andre enheter ved bruk av kommandoer. Bruk av attributter og kommandoer som sendes på denne måten beskrives som en KVP-melding. Navnet kommer fra selve dataverdien som betegnes som et *key value pair*.

Clustere kan inneholde en samling av attributtene beskrevet ovenfor, eller ingen i det hele tatt. Dersom ingen attributter er definert innenfor en cluster må applikajsonsdesigneren selv bestemme hvordan data og informasjon skal struktureres i meldingen. Vi har da en MSG-melding. Som nevnt innledningsvis benyttes også clustere ved såkalt *binding*. Kort forklart foregår dette ved at enheter sender en binding request til koordinatoren som vedlikeholder en bindingtabell for nettverket. Her *matches* inn og ut clustere som følge av den unike id'en de deler innenfor samme profil. Etter at denne bindingen har skjedd kan enheter kommunisere uten å spesifisere mottaker adresser.

Er det fler enn en underenhet i en node må vi ha en mulighet for å kunne adressere disse. I ZigBee er dette løst ved å innføre et endepunkt per underenhet. Ved bruk av endepunkter blir det mulig å ha opptil 240 underenheter per node.

Figur 4.2: Eksempel*device* med clustere og atributter.

Figur 4.2 viser et skjermbilde hentet fra ProfileBuilder, et program for profilutvikling medfølgende ZigBee-*stacken*. Prosjektet som er åpnet på bildet er et eksempel på oppbygningen av en enhet; *SafetyDevice*. Denne tilhører en bestemt profil ID som er spesifisert under egenskaper for enheten. Vi ser to clustere navngitt *SmokeSensor* og *HouseLights*, mens de underliggende instansene er attributter. I dette eksempelet blir de to clusterne sett på som to forskjellige underenheter med medfølgende definerte endepunkter for adressering. Dette er skjult på bildet da informasjonen befinner seg under *device*egenskapene.

Kapittel 5

HART - Highway Addressable Remote Transducer

5.1 Historisk overblikk

HART (*Highway Addressable Remote Transducer*) er en åpen nettverksprotokoll utviklet for digital kommunikasjon mot smarte felt enheter. Det var Fisher Rosemont (FR) som kom frem til denne protokollen under utvikling av enheter som krevde ekstra informasjon og funsjonalitet. Løsningen de til slutt endte opp med på starten av 1980 tallet var å lagre digital informasjon over et eksisterende analogt signal. Det ble raskt klart at en slik teknologi var noe 4-20mA standarden ville kunne dra stor nytte av. FR lanserte nyvinningen åpen for industrien hvorpå fremtidige enhetsdesign kunne baseres.

Opprinnelig var HART ment som et konfigurasjonsverktøy for justering av grenseverdier og andre viktige parametere for analoge sensorer. Men det ble raskt innsett at HART hadde fler muligheter og bruksområder enn kun som konfigurasjons og feilsøkingsverktøy. Ved å kontinuerlig motta data kan HART også være med å øke sikkerheten og begrense nedetid. Det henvises til seksjonen om sikkerhet for mer informasjon.

Da HART kom på markedet på 1980tallet var den en revolusjon av mange grunner. Først bør det nevnes at dette var den første digitale kommunikasjonsformen mot industrisensorer som muliggjorde overføringer uten å forstyrret det analoge signalet.[1] Mange sensorer er plassert på ufremkommelige og utilgjengelige steder. At det ikke lenger trengtes å bli foretatt kalibreringsendringer ved enhetens fysiske plassering sparte derfor feltingeniørene for mye arbeid. En annen grunn til at HART har blitt såpass utbredt er at den muligjør en smertefri overgang fra eksisterende 4-20ma løsninger. En 4-20mA enhet kan enkelt skiftes ut med en HART-kompatibel enhet. Alle eksisterende kablinger og tilkoblinger kan benyttes. Per dato gjør dette fortsatt HART til et av førstevalgene for mange bedrifter når oppgradering til smarte instrumenter vurderes.

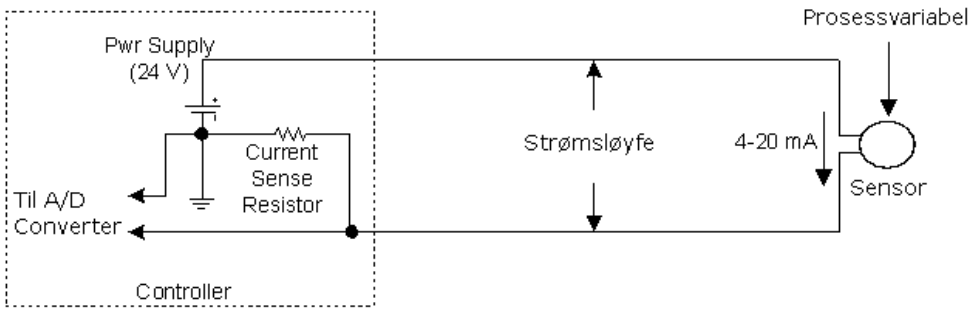
Idag er det HART Communication Foundation (HCF) som står for organisering

[1] Fortsatt er dette en unik egenskap som kjennetegner HART

og promotering av HART. På deres hjemmesider [6] kan det leses at det idag er over 150 bedrifter som støtter mer enn 800 forskjellige registrerte HART-enheter.

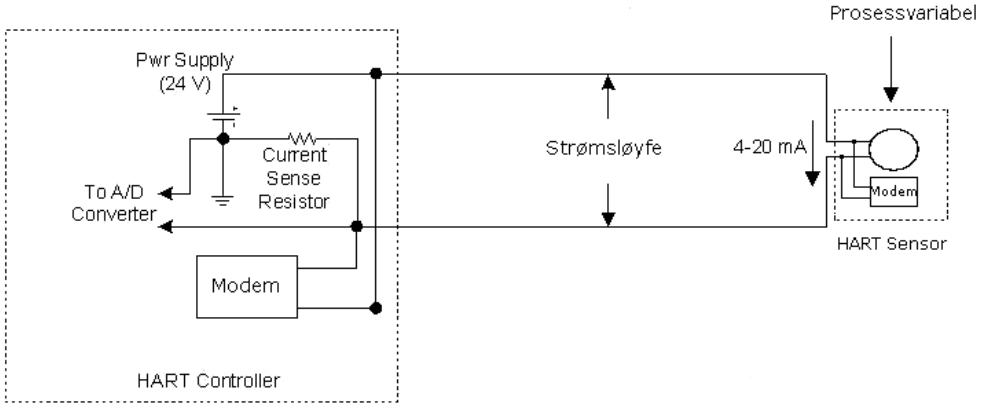
5.2 Oppbygning og Prinsipp

Kommunikasjon foregår mellom to HART-kompatible enheter. Vanligvis en feltenhet (sensor, reciever osv) og en *controller* eller monitoreringssystem (PC, DCS, Remote I/O osv). For å forstå oppbygnen til HART kan det være greit å begynne med dens opprinnelse. I *figur 5.1*, modifisert fra [8] ser vi hvordan en standard 4-20 mA sløyfe er bygd opp. Det er kun en enhet per sløyfe. Sensoren sender signal ved å variere strømmen som går gjennom den. *Controlleren* leser denne strømvariasjonen ved å måle spenning over en strømfølende motstand (vanligvis 250Ω). Som navnet tilsier varier strømmen mellom 4 og 20 mA. Frekvensen er vanligvis under 10Hz.



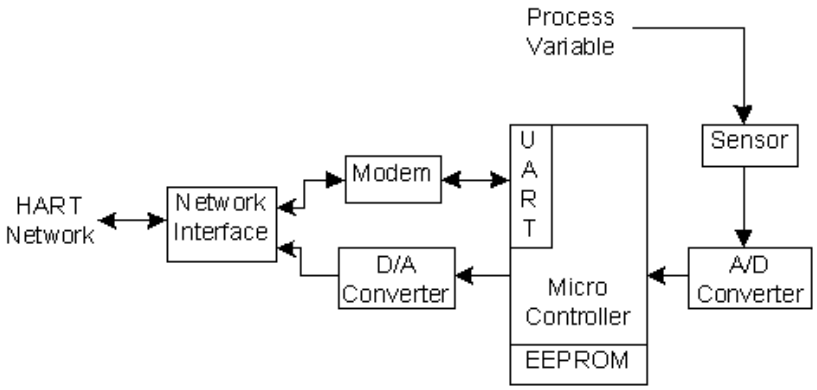
Figur 5.1: 4-20 mA strømsløyfe.

På det neste bildet, *figur 5.2*, er strømløyfen bygd ut til støtte HART-meldinger. Det varierer hva slags deler som er innebygd i forskjellige HART-modem. Det er på bildet valgt å skjule alle komponenter da det meste idag er integrert. De oppgavene modemet utfører er å modulere og demodulere meldinger som henholdsvis sendes og mottas. Moduleringen foregår ved hjelp av en teknikk som beskrives i neste underseksjon, FSK. Det er denne teknikken som gjør det mulig å sende og motta HART-meldinger samtidig med det analoge signalet. En nærmere beskrivelse av hva som foregår her kan finnes i [8], *Part 1.*



Figur 5.2: 4-20 mA HART strømsløyfe.

Ettersom innkommende signal blir oversatt til vanlige databit av demodulatoren sendes de til enhetens prosesseringsenhet via UART. Denne overføringen foregår med en hastighet på 1200 baud. En beskjeden hastighet sammenlignet med andre digitale kommunikasjonsnettverk som Feltbus eller Profibus, hvor hastighetene oppgis i megabit per sekund[2]. På mange områder vil allikevel HART være å foretrekke. Lave kostnader, lav kompleksitet og samtidig overføring med analogt signal er bare noen av fordelene. På *figur 5.3* ser vi oppbygningen av en typisk HART-enhet, hentet fra [8].



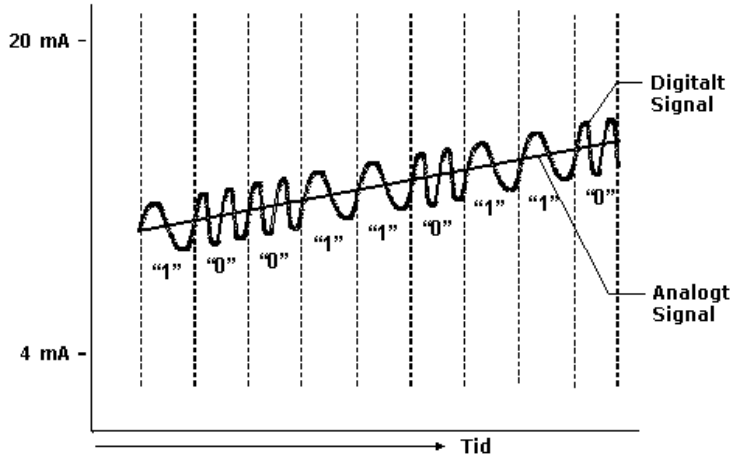
Figur 5.3: Oppbygning av en typisk HART-enhet.

FSK - *Frequency Shift Keying*

Kommunikasjonsprotokollen til HART er basert på *frequency shift keying* (FSK) prinsippet. Det digitale signalet er laget ved hjelp av to frekvenser, 1200 Hz og

[2]De høye hastighetene gjelder for kontrollbussene, men det er fortsatt mulig å bruke disse direkte mot sensorer. Utgavene for feltbuss har hastigheter definert til 31,25 kb/s.

2200 Hz som representerer bit 1 og 0. Det benyttes kontinuerlig FSK, noe som betyr at det ikke vil oppstå diskontinuitet på modulatorutgangen når frekvensen endres. Sinusbølger av de to frekvensene blir lagt sammen med det analoge signalet for å få samtidig analog og digital kommunikasjon. Fordi gjennomsnittsverdien av FSK signalet alltid er null vil ikke det analoge 4-20mA signalet bli påvirket. *Figur 5.4*, modifisert fra HCFs "*HART basic course*" [7], illustrerer prinsippet. En minimums sløyfeimpedanse på 230 ohm er nødvendig for kommunikasjon.



Figur 5.4: Prinsipp for FSK modulert signal.

5.3 Kommunikasjonsmodi

Det er to tilgjengelige kommunikasjonsmodi i HART-teknologien: *Request-Response Mode* og *Burst Mode*

Request - Response Mode

Denne modien fungerer slik at en vertsenhet, ofte kalt for en *master*, sender en forespørsel til den enheten den ønsker å kommunisere med. To *mastere* kan kobles til hver HART-sløyfe. Den primære *masteren* er som oftest et distribuert kontrollsystem (DCS), programmerbar logisk kontroller (PLC) eller en datamaskin (PC) som kjører en applikasjon. Den sekundære *masteren* kan være en håndholdt terminal eller en annen PC med en egnet HART-kompatibel applikasjon. HART-kompatible enheter inkluderer følere, aktuatorer, strømningsmålere, analysatorer, ventilposisjonerere og kontrollere som utfører kommandoer fra primær eller sekundær *master*.

Burst Mode

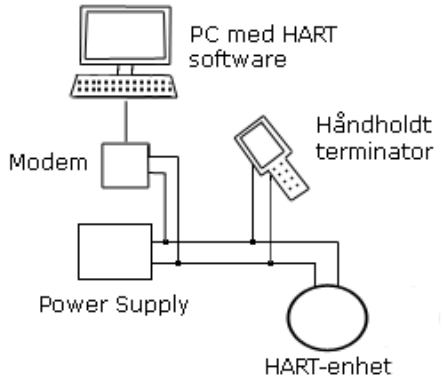
Burst mode muliggjør hurtigere kommunikasjon (3-4 data oppdateringer per sekund). I denne modien instruerer *masteren* en enhet til å kontinuerlig kringkaste en standard HART svarmelding (f.eks. PCV). *Masteren* mottar meldingen i høy hastighet inntil den ber enheten om å stoppe utsendingen.

5.4 Nettverkskonfigurasjoner

HART-enheter kan operere i en av to nettverkskonfigurasjoner, *point-to-point* eller *multidrop*.

Point To Point

I *point to point* oppkobling benyttes det tradisjonelle 4-20 mA signalet til å sende PCV, mens andre prosessvariable, konfigurasjons parametere og annen enhetsdata sendes via det digitale signalet. Det digitale HART-signalet påvirker ikke det analoge 4-20 mA signalet og kan benyttes til kontroll på vanlig måte. I *figur 5.5* er sløyfen illustrert. Her er det også tegnet inn en sekundær *master*, en håndholdt terminal som kan benyttes til testing og konfigurasjon. Sløyfemotstanden er ikke vist på bildet.



Figur 5.5: *Point to Point* oppkobling.

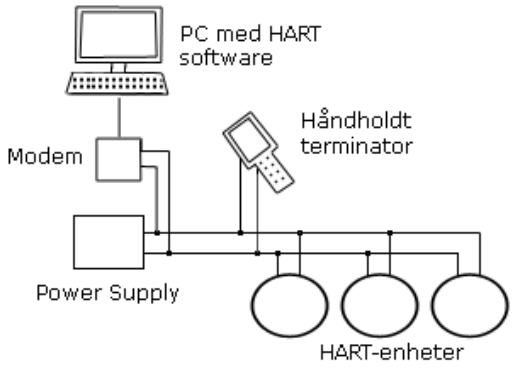
Multidrop

I *multidrop* oppkobling benyttes kun det digitale signalet. Opptil 15 enheter kan kobles sammen ved å bruke et enkelt ledningspar. I tillegg trengs det en ekstern strømforsyning og om nødvendig, sikkerhetsbarrierer. Alle feltenheter blir tildelt en pollingsadresse og strøm gjennom hver enhet blir stilt til et minimum (vanligvis 4mA).

Denne oppkoblingen benyttes ofte ved overvåkningskontroll og for innstallasjoner som er spredt over store områder. Noen eksempler kan være rørledninger, tankanlegg og oppbevaringsstasjoner.

Tiden det tar å hente ut informasjon om en enkelt variabel fra en HART-enhet er ca 500 ms. For et nettverk på 15 enheter vil det da ta 7,5 sekunder for å kontakte og hente ut de primære variablene fra alle enhetene. Skal det hentes informasjon fra multivariable enheter kan det ta ennå lenger tid siden datafeltet da vil inneholde data fra flere variable. Det bør altså vurderes hvor hurtig system som trengs før denne konfigurasjonen velges.

På *figur 5.6* vises det hvordan en slik oppkobling kan se ut. Her er det tegnet inn både primær og sekundær *master* i form av PC og håndholdt terminal. Sløyfemotstanden er ikke vist.



Figur 5.6: *Multidrop* oppkobling.

5.5 HART Kommandoer

Kommandosettet til HART spesifiserer enhetlig og fast kommunikasjon for alle HART-kompatible enheter. Kommandosettet er delt inn i tre klasser, universelle (*universal*), vanlig praksis (*Common Practice*) og enhetsspesifikke (*Device Specific*). Mellom 35 og 40 dataelementer er standard i enhver HART-registert enhet.

Universelle

Alle enheter som benytter HART-protokollen må gjenkjenne og støtte de universelle kommandoene. Disse gir aksess til nyttig informasjon under normal drift (f.eks. lese primær variabel og enheter).

Vanlig Praksis

Kommandoer som støttes av de fleste, men ikke nødvendigvis alle enheter. HART-spesifikasjonen rekommenderer at enheter støtter disse om de er anvendelige.

Enhetsspesifikke

Kommandoer som er unike for hver enkelt enhet. Disse kommandoene aksesserer oppsett og kalibreringsinformasjon samt informasjon om enhetsoppbygning. Informasjon om enhetsspesifikke kommandoer er tilgjengelig fra enhetsfabrikanter eller i enhentens medfølgende dokumentasjon (*Field Device Specification*).

5.6 *Device Description*

HART-kompatible enheter kan inneholde kommandoer og funksjonalitet som er unik for den enkelte enhet. En måte å få enkel tilgang til denne funksjonaliteten er hvis det er skrevet en *Device Description* (DD) for enheten. Denne filen inneholder en beskrivelse av variabler, kommandoer, menyer og skjermformater. Disse er skrevet i et spesielt språk, *Device Description Language* (DDL). Ved å kompilere og bruke DD filen i vertsapplikasjonen din vil vi kunne aksessere alle parametere og data enheten har å tilby.

Det som har skjedd på denne fronten i det siste er en oppgradering av DDL spesifikajsonen for å støtte avansert datavisualisering (eDDL). Blant annet i form av fullskjerm grafiske *display* og dataavlesning for ytelsesvisning. Dette tillegget i spesifikajsonen er gjort i et samarbeid mellom HCF, Fieldbus Foundation og Profibus.

5.7 Signaler og meldinger

Signalgang

Hart benytter seg som sagt av UART mellom prosessorenheten og modulator/demodulatoren. UART tar for seg sending av en byte av gangen. Før den sender en byte til modulatoren legger den til et startbit, et stoppbit, samt et paritetsbit for feilsjekking i mottaker. Disse tre ekstra *bit'ene* er med å bidra til overhead i HART-kommunikasjonen.

Før modulatoren sender de FSK modulerte bitene videre til nettverket aktiverer den et bæresignal, heretter kalt *carrier*. Dette er er også en måte å si ifra til mottageren om innkommende melding. Meldingen sendes så ut på nettverket. Mottager tar imot, demodulerer og konverterer de 11 *bit'ene* tilbake til en byte. På dette tidspunktet sjekkes også paritet for å se om meldingen ble riktig mottatt. Slik fortsetter kommunikasjonen byte for byte helt til alle bytene i en melding er sendt. Sender avslutter kommunikasjonen ved å sette *carrier* lav igjen.

Det er mange detaljer rundt signalgangen som er viktig å ta hensyn til når det skal sendes en melding. For eksempel kan det oppstå en del forstyrrelser i starten av en melding som følge av at *carrier* settes. Dette løses ved å sende et definert antall preamble bytes i starten av meldingen. Andre situasjoner som kan oppstå er forekomster av *gaps* og *dribbles*. Dette er henholdsvis når en slave ikke klarer å følge opp baudraten på 1200 bits per sekund, og når det oppstår såkalte "fan-

tombyte" i slutten av en melding (etter CRC byte). Den førstnevnte er sjelden et problem i nyere UART kretser, mens problemet med "fantombyte" må sjekkes av enheten for å se om det er søppel eller starten på en ny melding. Mer detaljer rundt dette kan finnes på [8]. Her kan det også leses hvordan HART-protokollen har løst synkronisering og timingspørsmål. Det er valgt å kun gi en minimal forklaring på disse tingene da det ligger utenfor det området som er tema for denne oppgaven.

Timing er viktig om flere *mastere* kobler seg til samme sløyfe. HART tillater to *mastere* å være tilkoblet samtidig. Det dreier seg da som oftest om en kontrollenhet (PC, DCS osv.) og en håndholdt terminal. Når to *mastere* har meldinger å sende kan det lett oppstå kollisjoner. Dette er løst ved at hver *master* må vente et forhåndsdefinert tidsrom etter en meldingssending og mottak fra slave, før den kan sende på nytt. I dette tidsrommet kan den andre *masteren* starte en meldingssending. Dersom begge *masterne* forsøker å sende samtidig vil de begge bli satt til å vente et bestemt tidsrom før de på nytt kan sende. Dette tidsrommet vil være forskjellig for de to *masterne* slik at de pånytt ikke vil starte samtidig. Flere scenarioer kan tenkes her, hvorav alle er løst ved timingmetoder. Mer om dette kan som sagt leses i overnevnte referanse.

Meldingsoppbygning

Under normale omstendigheter er det kun en enhet i HART-nettverket som snakker av gangen. Kommunikasjonen foregår ved at en *master* sender en kommando til en slaveenhet og venter på svar. Slaven på sin side venter på en kommando som den deretter svarer på. En kommando med tilhørende svar kalles en transaksjon.

Delene til en typisk HART-melding er beskrevet nedenfor. De er gitt i den rekkefølgen de ville forekomme i meldingen.

Preamble: benyttes for å synkronisere enhetene og forsikre seg mot støy i starten av meldingssendingen. Lengden av denne kan variere fra 5 til 25 bytes, inneholdende FF i hex. Ved førstegangskommunikasjon med en enhet benyttes maksimum antall *preamblebytes*. Når *master* har lest slavens preamblelengde kan denne benyttes, vanligvis er denne på 5 byte.

Start: består av en byte som forteller oss hva slags adresseformat og meldingstype vi har. Meldingstyper kan være *master* til slave, slave til *master* eller *burst melding*. Adresseformatet kan være lang eller kort adresse (4 eller 38 bit).

Adresse: inneholder både *master*-adresse og slaveadresse. *Master*-adressen er et enkelt bit som indikerer primær *master* ved 0, og sekundær *master* når den er 1. Når vi har kort adresseformat, (*short frame*), er slaveadressen på 4bit, inneholdende polleadressen (0 til 15). I det lange adresseformatet, *long frame*, inneholder den enhetens unike adresse på 38 bit. Et bit benyttes også til å fortelle oss om slaven er i *burst mode*.

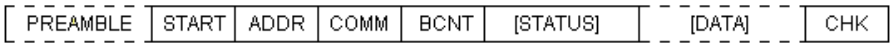
Command: forteller slaven hva den skal gjøre. Kommandoen kan være av en av de tre typene nevnt under seksjonen "HART-Kommandoer". Ønskes det å sende en *device specific* kommando kan det være nødvendig å oppgradere *master*-enheten med en DD fil slik at disse blir tilgjengelige.

Byte Count: er en byte som forteller hvor mange bytes som følger i *status* og *data bytes*. Mottakeren benytter denne informasjonen for å vite når meldingen avsluttes.

Status: benyttes kun ved svar fra slave. Denne gir oss informasjon om slavens status og eventuelle kommunikasjonsfeil mm.

Data: har forskjellig lengde avhengig av meldingstype og kommando som skal utføres. Kan variere mellom alt fra lange meldinger til ingenting.

Checksum: inneholder en eksklusiv-*or* eller "*longitudinal parity*"av alle byte fra startkarakteren og fremover. Sammen med paritetsbitet til hver byte benyttes dette for å detektere kommunikasjonsfeil.



Figur 5.7: HART meldingsoppbygning.

5.8 Sikkerhet

Intelligent data er som nevnt ikke bare tilgjengelig gjennom de håndholdte kommunikatorene eller et lignende program. Dataene kan monitoreres kontinuerlig 24 timer i døgnet, 7 dager i uken.

Gjennom kontinuerlig monitorering av det digitale kommunikasjonssignalet kan det lett detekteres potensielle feil i kontroll system på enhetsnivå. HART-detekterbare feilscenarioer inkluderer sensor feil (utenfor grenseverdier), 4-20mA forstyrrelse (analog utgang er ulik PCV måling), feil oppsett eller kalibreringsfeil i enheten (enhetens variasjonsbredde er ulik variasjonsbredden til kontrollsystemet)

Kontinuerlig overvåkning av HART-data kan være kritisk for anleggsoperasjoner. Ifølge en artikkel utgitt av Chemical Prosessing [12] ble nylig et stort amerikansk selskap rammet av en tre-dagers nedstengning på grunn av en feilende nivåsensor hvor feilen ikke ble detektert. En undersøkelse viste at vann hadde trengt inn i nivåsensoren og delvis kortsluttet den analoge utgangen. Dette resulterte i at feilaktige nivåmålinger ble mottatt i kontrollrommet. Hadde kontrollsystemet monitorert HART-kommunikasjonssignalene ville feilen blitt detektert i løpet av sekunder, ved å sammenligne verdien av den digitale PCV med den analoge. Med en ulikhet her ville systemet kunne alarmert operatøren og potensielt kunnet spare fabrikken for over $300,000 i nedetid.

Ved å benytte kontinuerlig monitorering i et HART-kompatibelt system kan det i tillegg til å detektere feil også kunne detekteres andre abnormaliteter. Dette kan være hendelser som at enhetsoppsettet er endret, en selvtest kjører eller en reset har blitt utført i feltet.

5.9 HART 6

HART-enheter har med årene blitt mer avanserte og inneholder stadig mer funksjonalitet. For å holde tritt med denne utviklingen har vi idag nådd protokollversjon HART 6. Denne er fullt og helt bakoverkompatibelt med tidligere versjoner. For brukere som har utstyr og systemer tilhørende eldre protokollversjoner vil altså disse være fullt kompatible med HART 6 produkter. De vil derimot ikke få utnyttet den nye funksjonaliteten i den nye protokollen. Når en HART 6 enhet kobles til en HART 5 *master* vil den oppføre seg som en HART 5 enhet. Noen av største endringene fra HART 5 til HART 6 er oppsummert nedenfor:

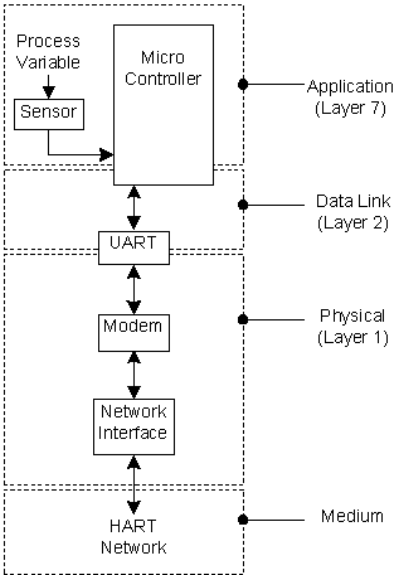
- Forbedret integrering i vert ved hjelp av *Device Descriptin* (DD).
- Forlenget data og statusinformasjon per melding.
- *Tag* er forlenget til 32 karakterer
- Oppdatert *Block Data Transfer* for overføring av store mengder data.
- Nye *common practice* kommandoer for å støtte spørring og feilsøking for enheter koblet i *multidrop* og multikablede installasjoner.

5.10 OSI modellen for HART

HART implementerer tre av lagene i OSI modellen, det fysiske laget, datalinklaget og applikasjonslaget. Det fysiske laget står for følgende funksjonalitet:

- Modulering av utgående meldinger
- Demodulering av innkommende meldinger
- Skru på *carrier* (bæresignal) for utgående melding.
- Detektere *carrier* for innkommende melding

Det går ikke an å dele inn HART inn i de forskjellige lagene på detaljnivå. UART'en kan for eksempel sies å være en del av både det fysiske laget og datalinklaget. Det fysiske laget da den er ansvarlig for å å lage en strøm av bit, mens den også legger til paritetsbit for feilkontrollering. Dette er en funksjon som vanligvis tilhører datalinklaget. Grovt sett vil OSI modellen se ut som på *figur 5.8*, hentet fra [8], *Part 3*.



Figur 5.8: OSI modellen for HART.

Del II

Implementering

Kapittel 6

Utstyr og fysisk implementering

6.1 Utgangspunkt for arbeidet

For å gjennomføre prosjektet var det nødvendig med to ZigBee-noder med tilhørende *stack*, en HART-kompatibel sensor og et grensesnitt for å kommunisere med denne. Fra tidligere prosjekter hadde Statoil deler av et utviklingssett liggende ved forskningssenteret på Rotvoll. Settet var opprinnelig innkjøpt av ABB fra det norske selskapet Chipcon og inneholdt flere utviklingskort, samt Figure 8 Wireless sin implementering av ZigBee-*stacken*. Utviklingskortene var av typen CC2420DB, inneholdende Atmega128 mikrokontroller fra Atmel, Chipcons CC2420 radiobrikke og enkel I/O. I tillegg hadde kortene også seriegrensesnitt med flytkontroll for interaksjon mot datamaskin og andre vertsenheter. Versjonen av ZigBee-*stacken* som ble benyttet var frigivelse 1.0, versjon 1.1.0. Sammen utgjør utviklingskort og *stack* en god plattform og startpunkt for utvikling av ZigBee-løsninger.

I forbindelse med tidligere prosjekter var det bygd en kombinert demonstrasjons- og opplæringsrigg[1] ved laboratoriet på Rotvoll. Denne består av en strømningssløyfe med to tanker, akkumulator og varmeelement. Riggen er ellers instrumentert med trykksensor, ultralyd nivåmåler, flowmeter og temperatursensor. Som følge av HART-prosjektet i denne rapporten ble det besluttet å oppgradere riggen med en HART-kompatibel trykksensor. En Rosemount 3051 trykksensor ble innkjøpt fra Emerson, mens oppgraderingen ble utført av lærlingene Linda Wrangen og Anja Engtrø. Det var også nylig gjort innkjøp av en 375 håndholdt terminal, denne kom godt med under arbeidet med prosjektet. Bilder av utstyr og oppkobling kan finnes i *vedlegg A*.

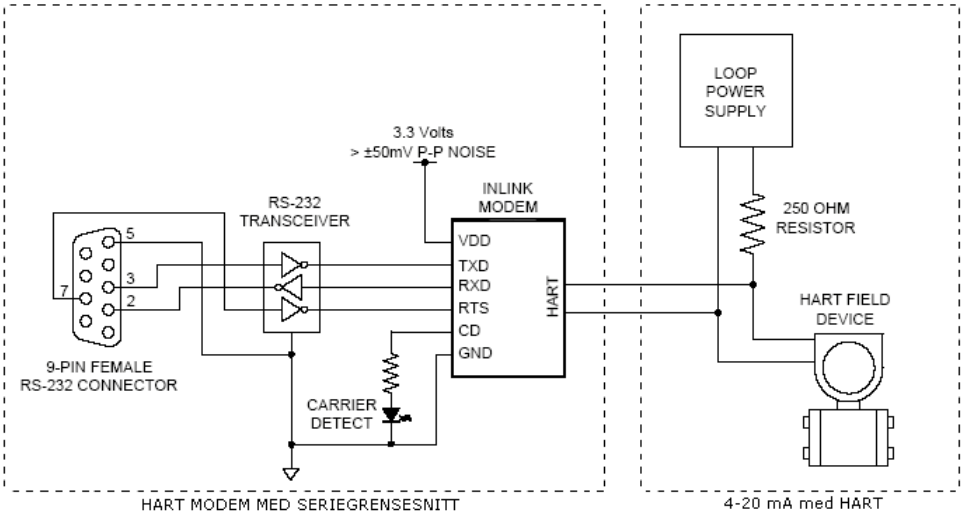
6.2 Grensesnitt mot HART-sensor

Tenkt design

Et av punktene for gjennomføringen av dette prosjektet var å se på et grensesnitt mellom ZigBee og HART. Da arbeidet skulle til å begynne ble det forøvrig

[1] Opprinnelig designet av Gisle H. Bedin og bygd av lærlingene Linda Wrangen og Eivind Utheim.

bestemt at et HART-modem med seriegrensesnitt ville bli innkjøpt. Før dette ble avklart ble det allikevel sett litt på mulighetene for å lage dette selv. Et kjapt søk i google[16] vil gi mange treff på ulike leverandører som leverer de nødvendige komponentene. Det dreier seg her om integrerte kretser som tar seg av frekvensmodulering og sammensetning av bits til bytes. Brikkene har innebygd uart, mens konvertering til RS232 ikke nødvendigvis behøver å være integrert. Et godt eksempel på en slik brikke ble funnet på nettsidene til Microflex[17] som også er leverandør av annet HART-relatert utstyr. De leverer en kompakt brikke som enkelt kunne hvert implementert i de fleste mikrokontroller-design. For å oppnå samme funksjonalitet som innkjøpte HART-modem måtte det i tillegg bli kjøpt inn en RS232-koverter og nødvendige tilkoblingpunkter for seriekabel og HART. Konverteren kunne for eksempel vært en MAX232-brikke som kan bestilles gjennom Elfa[18], 9 pins D-SUB for RS232 og andre tilkoblinger kan også finnes her. I *figur 6.1* er det vist hvordan kretsen kan kobles opp. Bildet er modifisert fra brikkens datablad.



Figur 6.1: Oppbygning av krets rundt integrert modembrikke.

Innkjøpt HART-modem

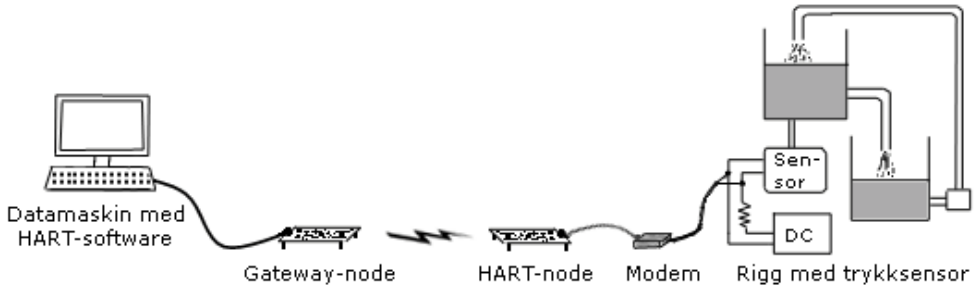
Modemet som ble kjøpt inn til prosjektet var av modelltype HI311, produsert av *Smar Research Corporation*. Til felles med designet i forrige avsnitt har også dette modemet seriegrensesnitt for tilkobling mot datamaskin. Siden både modem og ZigBee-node er laget for å kommunisere mot en *master* ble det nødvendig å benytte en krysset seriekabel[2] mellom enhetene. Kontroll av modemet foregår ved RTS-linjen i seriegrensesnittet. Høy RTS tillater sending til sensor, mens lav gir tillatelse til å motta fra sensor. Databladet forteller også at modemet trekker driftspenning fra seriekabelens DTR-linje. Dersom denne er lav var det

[2]Ordet nullmodemkabel kunne vært brukt, men dette navnet beskriver egentlig en sammenkobling hvor det ikke finnes noe modem.

beskrevet at strøm heller ville trekkes fra RTS og TxD. Det betyr altså at dersom ikke DTR-linjen settes høy vil modemet kun fungere ved utsending av data. I ZigBee-noden er ikke DTR-linjen i bruk og det ble derfor nødvendig å benytte en ekstern strømforsyning for modemet.

6.3 Testoppsett

En stegvis oppkobling av utstyret i forrige avsnitt resulterte til slutt i et endelig og virkende testoppsett som vist i *figur 6.2*. Første steg var derimot kun bruk av modemet koblet mellom sensor og datamaskin for direkte kommunikasjon. Henviser til seksjonen 'Driver for HART' i neste kapittel for mer informasjon om dette. Når denne sammenkoblingen virket ble neste skritt å få ZigBee-nodene til kommunisere. I første omgang ble nodene koblet til hver sin PC via RS232 hvor terminalvinduer ble brukt for å oversende tekststrenger. Deretter ble de forskjellige delene satt sammen og den endelige testingen begynte. Problemstillinger og design vil bli diskutert i neste kapittel.



Figur 6.2: Endelig testoppsett for prosjektet.

Kapittel 7

Applikasjonsdesign

I dette kapittelet vil valg av designløsninger og implementering bli diskutert. I hovedsak dreier det seg om valg av programvare på datamaskinen samt utvikling av applikasjonen for ZigBee-enhetene. Disse oppgavene er nært knyttet til hverandre da de må snakke samme språk for å sikre feilfri kommunikasjon. I starten av neste seksjon vil valget av løsning bli diskutert, før fokus rettes mot selve implementeringen. Til sist vil det bli sett litt på utvalget av ulike gratis programmer som kan benyttes mot HART.

7.1 Driver for HART

For kommunikasjon mot HART-modemet ble flere muligheter vurdert. Etter å ha studert meldingsoppbygningen til HART ble det først vurdert å lage en driver direkte i ZigBee-noden som skal kommunisere med modemet. Denne fremgangsmåten vil skape minimalt *overhead* ved overføringer mellom ZigBee-nodene som følge av at den spesifikke delen av HART-meldingen vi vil oversende kan plukke ut. Risikoen for timingproblemer vil også reduseres da svar fra modemet (og motsatt) vil behandles momentant. Med timingproblemer menes her den lovlige tiden et program eller enhet venter på svar før ny polling etter data. Dette gjelder kun i sammenheng med programvare for HART hvor disse innstillingene ikke er redigerbare. I tillegg til driveren i ZigBee-noden ville det selvsagt også være nødvendig med programvare på datamaskinen for mottak og visning av data, samt valg av kommando som skal sendes. Det enkleste alternativet her ville vært å bruke et vanlig terminalvindu ved testing og heller bygge på med et mer avansert avlesningsprogram når resten av kommunikasjonen fungerer feilfritt.

En annen mulighet ville være å benytte ZigBee som en grå kanal mellom datamaskinen og HART-modemet. Dette betyr at programvaren på datamaskinen vil kunne kommunisere med modemet som om de var direkte sammenkoblet. Ved denne fremgangsmåten vil ZigBee-enheter kunne kobles inn i et system uten at eksisterende programmer trenger å bli utskiftet. Dette ble til slutt vurdert som så fordelaktig at valget falt på denne løsningen.

Få sensoren til å snakke

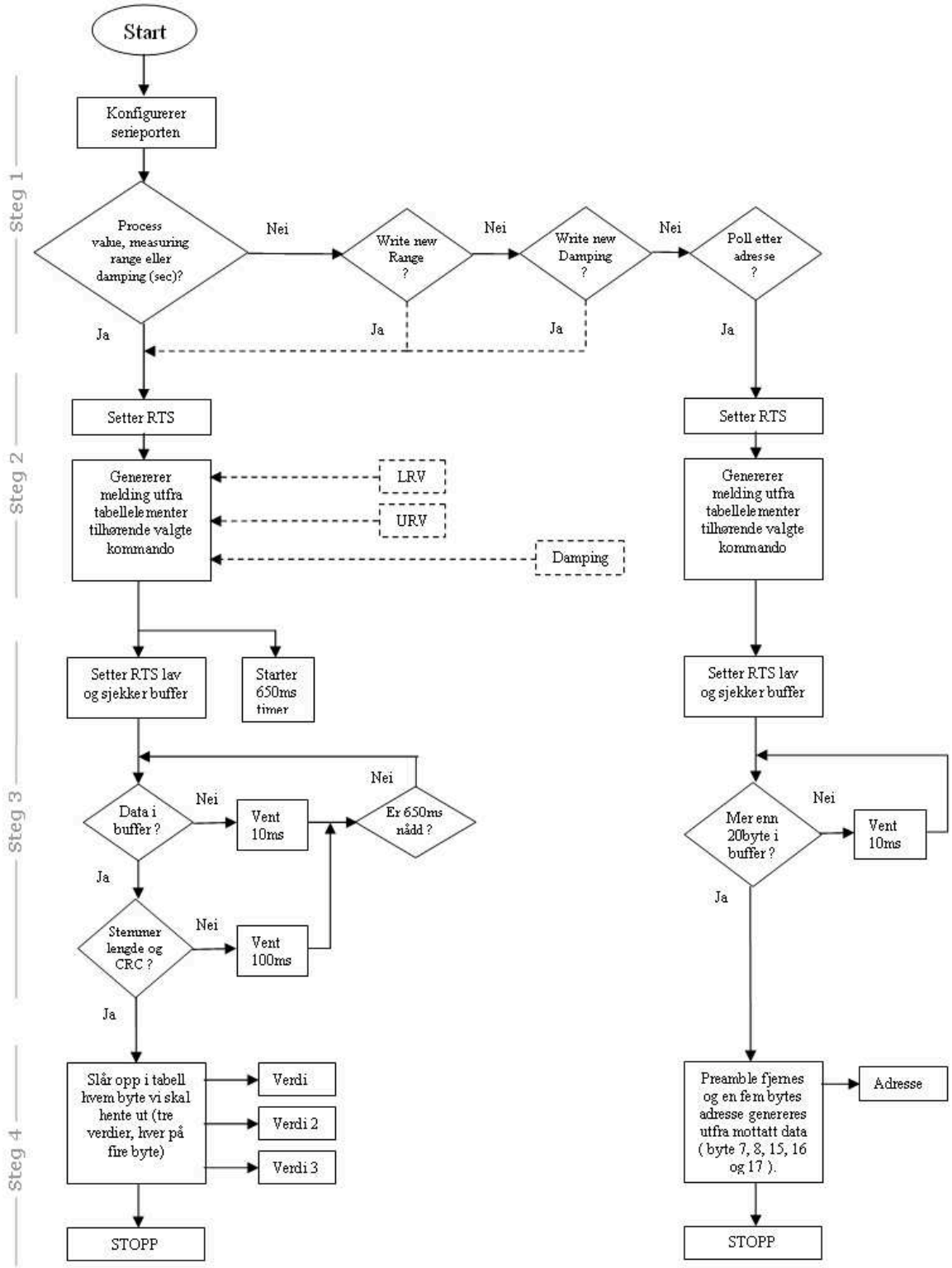
Naturlig nok er det første skrittet i en slik prosess å få kontakt med HART-modem og sensor gjennom programvaren vi har tilgjengelig. Flere gratis kommunikasjonsprogram for HART ble prøvd uten at det gav noe resultat. Til sist ble det prøvd med en driver skrevet for labVIEW[1]. Denne gav heller ikke et positivt resultat i første omgang, men siden det her var tilgjengelig kildekoden[2] gav det mulighet for utvidet testing i forhold til gratisprogrammene. Det første som må gjøres når det skal kommuniseres med en ny sensor er å polle etter sensorens adresse. Denne adressen kan så benyttes ved videre kommunikasjon. Ved noen enkle modifikasjoner i driveren for LabVIEW ble det mulig å se på den faktiske dataen som ble mottatt ved polling etter adresse. Ut i fra dette ble det mulig å trekke noen konklusjoner. Data ble faktisk mottatt fra sensoren, men det var feil i meldingen. Dette gjorde at meldingen ble forkastet av klommunikasjonsprogrammene. Nå som meldingen var tilgjengelig kunne en adresse genereres[3] ved å plukke ut de fem bytene i meldingen som utgjør enhetens adresse. Denne oppdagelsen gjorde det nå mulig å prøve ut andre kommandoer enn adressepolling. Det viste seg at disse virket utmerket.

Driver for HART i labVIEW

Design og utarbeidelse av en driver for HART var ikke et av målene ved dette prosjektet. Det ble allikevel nødvendig å sette seg grundig inn i driveren som til sist ble brukt. Under arbeid med denne ble det savnet dokumentasjon og kommentarer til de ulike blokkene inne i selve driveren. For å gjøre det lettere for andre som skal bruke denne driveren eller har planer om å lage sin egen er det laget en detaljert gjennomgang av driverens oppbygning. Denne vil også være nyttig for folk som vil ha en dypere forståelse for hvordan oppbygning og sending av HART-meldinger foregår. Når det gjelder oppgradering av selve driveren er det kun gjort minimale endringer. Kun småpynting og kommentarer er lagt til.

Driveren er oppbygd slik at den vil kjøre helt til den mottar et CRC godkjent svar fra slave[4]. Den har nå kun lagt til seks kommandoer, men dette kan lett utvides etter ønske. Valg av kommando vi vil sende velges i en nedtrekksmeny. Hver av disse har et skjult nummer fra 0 til 5. Nummeret til en spesifikk kommando vil fortelle oss hvilken rad (eller kolonne) i tabellen som tilhører denne. *Vedlegg B.1* viser et bilde av brukergrensesnittet. De forskjellige tabellene og deres innhold kan leses ut i fra denne. I detaljene rundt blokkene (eller stegene) i driveren vil det heller henvise direkte til dataelementene som blir satt sammen. Det er ikke skilt mellom betydningen av forskjellige byte i tabellene. På neste side vil det bli gitt et blokkskjema over driverens virkemåte. De forskjellige stegene til høyre i skjemaet vil beskrives i påfølgende sider.

[1] En takk går her til Tom Odin Gaustad og Magne Sætre som har utviklet driveren.
[2] Kildekode er her i form av blokker og ledninger.
[3] Mer informasjon rundt dette kan finnes under detaljene for labVIEW-driveren.
[4] Fra kapittelet om HART.



Figur 7.1: Funksjonsskjema for labVIEW-driveren.

De forskjellige stegene avtegnet til høyre på blokkskjemaet vil ikke være nøyaktig like for alle kommandoer. Kommandobyten vil selvsagt være forskjellig, men for noen av kommandoene vil det også være andre vesentlige forskjeller.

Steg 1 - Konfigurasjon av serieport

Dette steget utføres likt uavhengig av kommandoen vi vil sende. Riktig konfigurasjon av serieporten er avgjørende for at kommunikasjonen skal kunne foregå uten feil. Legg særlig merke til det siste punktet da dette ofte utelatt i databladene.

- Portnummer: avhenger av tilkobling.[5]
- Baudrate: 1200 baud.
- Databits: 8 bit.
- Stoppbits: 1 bit.
- Paritet: Odde paritet.

I tillegg til overnevnte punkter krever også mange modem tilført driftspenning. Dette trekkes i mange tilfeller fra serielinjen ved at DTR-signalet settes høyt. RTS bør også settes høy her for å si ifra til modem at vi er klare til å sende data. Mot ZigBee-noden er dette ikke nødvendig.

Steg 2 - Generering og sending av melding

Generering og sending av melding gjøres litt forskjellig for de ulike kommandoene. Som vi skal se er det særlig adressepolling som skiller seg mest ut. For de tre første kommandoene; *process value*, *measuring range* og *damping(sec)* er det kun kommandobyten som utgjør forskjellen. Programoppbygningen er angitt punktvis nedenfor.

- Når denne blokken entres begynner en tidtaker å telle ned fra 130ms. Vi vil ikke gå videre til neste steg før denne har talt ned til null. Startverdi for nedtellingen må være høy nok til at utsending rekker å fullføre.
- RTS settes høy.
- Starter generering av melding. Det vil si at de forskjellige elementene den endelige meldingen vil bestå av flettes sammen. Det første elementet i meldingen er startbyte. Deretter hentes den fem byte lange adressen som ble funnet ved adressepolling. Etter denne er turen kommet til kommandobyten. I driveren inneholder ikke kommandotabellen kun kommandoer, men også elementet for bytecount[6] (BCNT) og påfølgende elementer. Tabellen kalt NR BYTES forteller oss hvor mange elementer vi skal ta med fra kommandotabellen. For de tre første kommandoene er dette kun de to bytene nevnt ovenfor.

[5] Vær oppmerksom på at nummerering starter på null i labVIEW.
[6] Se seksjon om meldingsoppbygning i kapittelet om HART.

- Neste skritt er nå å generere en sjekksum (CRC) for meldingen. Dette gjøres ved å foreta en ekslusiv eller operasjon på alle nylig genererte elementer. Denne CRC-byten legges så til som siste element i meldingen.
- Før utsending legges det også til preamble byte foran i meldingen. For vår sensor holder det med fire, men for eldre sensorer kan det være nødvendig med fler.
- Meldingen sendes ut på serieporten og vi går over til neste steg så snart tidtakeren fullfører nedtellingen.

I tabellen under vises oppbygningen av meldingen som vil sendes om process value er valgt. For de to andre meldingene vil kun kommandobyten (CMD) og CRC være annerledes.

Preamble				Start	Adresse					CMD	BCNT	CRC
0xFF	0xFF	0xFF	0xFF	0x82	0x26	0x06	0x21	0x77	0x86	0x03	0x00	0x71

Tabell 7.1: Melding for *process value*.

Kommandoene *write new range* og *write new damping* vil ha lik oppbygning som de tre første. Hovedforskjellen ligger i at vi her også vil sende med data inneholdende grenseverdiene eller dempningen vi vil sette. BCNT som var null for de tre første kommandoene vil nå inneholde en verdi som angir antall påfølgende databytes. Hver av verdiene vi vil sette vil oppta fire bytes. LabVIEW er genial på dette området da den direkte kan caste om verdier til et slikt format. CRC vil for disse kommandoene variere etter hvilke verdier som sendes og må selvsagt flettes inn til slutt. Under kan meldingsoppbygningen for de ulike kommandoene leses. Legg spesielt merke til *write new range* da det er denne som er årsaken til den ekstra tabellen kalt *commands old type*. Et ekstra element er her lagt til etter BCNT. Denne angir på nytt anntall kommende databytes og er forskjellig for nye og gamle sensorer. Det er blitt valgt å kalle elementet CNT.

Preamble				Start	Adresse					CMD	BCNT	CNT
0xFF	0xFF	0xFF	0xFF	0x82	0x26	0x06	0x21	0x77	0x86	0x23	0x09	0x08

DATA: URV				DATA: LRV				CRC
0x42	0xDC	0x00	0x00	0x41	0x20	0x00	0x00	0xAF

Tabell 7.2: Melding for *write new range*.

Preamble				Start	Adresse					CMD	BCNT
0xFF	0xFF	0xFF	0xFF	0x82	0x26	0x06	0x21	0x77	0x86	0x22	0x04

DATA: DAMPING				CRC
0x40	0x19	0x99	0x9A	0x0E

Tabell 7.3: Melding for *write new damping*.

Det gjenstår nå bare en kommandoen for å polle etter adresse. Her benyttes alle elementene i kommandotabellen, og det legges kun til preamble i start av melding før utsending. Den sendte meldingen vil se ut som beskrevet nedenfor.

Preamble				Start	Adresse	CMD	BCNT	CRC
0xFF	0xFF	0xFF	0xFF	0x02	0x00	0x00	0x00	0x02

Tabell 7.4: Melding for adressepolling (*poll adr*).

Steg 3 - Mottak av melding

Ved mottak av melding er det kun adressepolling som avviker fra de andre kommandoene. Det vil si at når vi i forrige steg pollet etter adresse vil dette medføre litt annen funksjonalitet i dette skrittet. De fem andre kommandoene vill bli tatt først.

- Setter RTS lav for å fortelle slave at vi er klare til å motta data.
- Starter en timer på 650 ms. Denne vil sende oss tilbake til steg 2 om vi ikke mottar CRC-godkjent melding fra slave innen *timeout*.
- Programmet vil nå gå i loop og sjekke buffer hvert tiende millisekund så lenge punktet ovenfor overholdes. Dersom noe blir funnet i buffer vil lengden på mottatt data sammenlignes med spesifisert meldingslengde for kommandoen, gitt i tabellen kalt *bytecount read new type*[7]. Ved riktig lengde vil CRC sjekkes, om lengden derimot er kortere enn forventet vil vi vente 100ms før vi pøny sjekker buffer. Dette gjøres for å forsikre oss om at all sensordata blir mottatt. Om lengde og CRC stemte i første forsøk vil vi gå videre til steg 4 for visning av data. Dette gjelder også når ny sjekk av buffer blir foretatt. Ved feil i CRC, eller feil lengde i andre forsøk vil vi gå tilbake til steg 2 for å polle etter ny data.

For adressepolling har steget en litt annen oppbygning. Det er her ingen form for CRC-sjekk eller timer som vil sende oss tilbake til steg 2 ved timeout. Ved å utelate denne funksjonaliteten vil programmet bli stående i en evig løkke om ikke riktig data blir mottatt i første forsøk. Dersom data blir mottatt vil det heller ikke kunne påvises at dataen er uten feil. Videre kan dette føre til generering av

[7]Se vedlagt skjermbilde av brukergrensesnitt for tabellinnhold

en adresse som ikke stemmer, noe som igjen medfører at all videre kommunikasjon vil være uten resultat. Det skal allikevel nevnes at adressen kan lagres når den først er mottatt. Altså trenger vi kun benytte denne kommandoen første gang vi skal kommunisere med en ny sensor. For ordens skyld er funksjonaliteten for steget listet opp i punktene under.

- Setter RTS lav for å fortelle slave at vi er klare til å motta data.
- Programmet går deretter i loop og sjekker bufferet hvert tiende millisekund. Dersom mottatt data overstiger 20 byte vil vi anta at riktig melding er mottatt og går til neste steg.

Steg 4 - Visning av mottatt data

Det fjerde og siste steget innebærer visning av mottatt data. Som i forrige steg er dette steget likt for alle kommandoer unntagen adressepolling. For denne kunne dette steget likegodt vært en del av steg 3. Etter at dataen er mottatt fjernes preamble, deretter genereres en 5 bytes adresse ut i fra gjenværende elementer. Dette innebærer fletting av element 7, 8, 15, 16 og 17 i den rekkefølgen de står skrevet[8]. Når adressen er generert vil den lagres i adressetabellen og kunne brukes ved videre kommunikasjon.

For de andre kommandoene vil vi foreta et oppslag i tabellen ved navn *indexing word read*. Ut i fra hvem kommando vi har valgt vil vi her kunne lese de tre startelementene for de tre avleste måleverdiene som hver har en lengde på fire byte. Avlest verdi vil så sendes ut på display og vises for brukeren før programmet stoppes.

Tabell 7.5 viser en melding mottatt ved polling etter prosessverdier. For å vite eksakt hva de ulike elementene representerer er det nødvendig å ha tilgang til protokollspesifikasjonen for HART[9]. Da det er selve kommunikasjonsdelen som er fokus for dette prosjektet er det valgt kun å peke ut dataelementene inneholdene pollet data. De fire preamblebytene er her tatt bort for å spare plass på siden.

Sensoradresse mm.	Value		Value 1		Value 2	
86 26 06 21 77 86 03 10 00 40	40 E3 C6 AA	08	41 80 2A EA	20	41 89 18 00	13 AA

Tabell 7.5: Melding (med fjernet preamble) mottatt fra sensor ved *process value*.

Dette vil omgjort tilsvare; Value: 7,1180, Value 1: 16,0210 og Value 2: 17,1367. Om dette er samme variable som for kommunikatorens meny for prosessvariable tilsvarer dette trykk, prosent av *range* og *analog output*.

[8] Denne kunnskapen gjorde det mulig å generere adressen da feil preamble ble mottatt.
[9] Spesifikasjonen kan skaffes gjennom [6].

7.2 Design av ZigBee-applikasjon

Som nevnt i starten av forrige seksjon ble det valgt å benytte ZigBee som kabelerstatning mellom *master* (her i form av en datamaskin) og modem. Dette kan høres ut som en enkel oppgave, men som det snart vil vises er det være mange hensyn å ta og mange problem å løse. Særlig er det kommunikasjonen mellom ZigBee-enhet og modem som kan skape endel hodebry.

For design av applikasjonen sto valget mellom å opprette en ny profil med *clustere* og endepunkt, eller programmere direkte mot ZDO[10] uten profil. Med tanke på videre utvidelser og muligheter falt valget til slutt på førstnevnte.

Utvikling av profil med ProfileBuilder

I utviklingspakken fra chipcon følger det med utviklingsverktøy for profilbygging. Denne har et enkelt og greit brukergrensesnitt, men mangler et grundig utviklingseksempel med forklaringer. For å kompansere for dette kommer det til å fokuseres ekstra på utvikling gjennom disse seksjonene.

Dersom det er ønskelig med forskjellig funksjonalitet i de ulike nodene må programmene lages hver for seg. For dette prosjektet er det valgt å lage en *gateway*-enhet og en HART-enhet hvor førstnevnte vil være koordinator, mens HART-enheten vil bli definert som en vanlig enhet.

Opprettelse og diskusjon av clusterparametere

Etter å ha startet opp ProfileBuilder kan clustere opprettes og manipuleres i vinduet med navn; *Cluster Workshop*. Under egenskapene for opprettede clustere kan det velges meldingstype, navn og clusterID.

I kapittelet om ZigBee ble forskjellen på KVP og MSG-meldinger nevnt. For å friske opp kan det nevnes at KVP-meldinger benyttes for å få tilgang til ulike lagrede verdier hos nodene. Hver lagrede verdi er assosiert med en attributtID som benyttes ved aksess. Tre kommandoer; *Set*, *Get* og *Event* benyttes for å manipulere verdiene. MSG er derimot et helt fritt rammeformat som applikasjonsdesigneren bestemmer selv. Dette formatet gir også mindre *overhead* som følge av kortere meldingsheader (2 byte) i forhold til KVP (4 byte). KVP har i tillegg en begrensning som sier at operasjoner kun kan utføres på standard datatyper. For å oversende en streng av bytes som er målet for dette prosjektet vil altså den klare vinneren være MSG-formatet.

Navnet på clusteren har ingen betydning for funksjon og virkemåte, men det bør brukes beskrivende navn som er lett gjenkjennelig da dette vil gjøre clusterne lettere å håndtere.

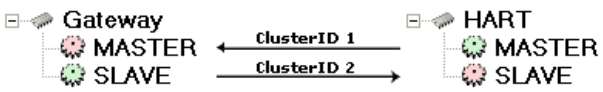
Til sist har vi clusterID som representerer entiter og funksjoner på en node.

[10] Dersom begreper som ZDO sitter løst anbefales det å lese igjennom kapittelet om ZigBee ennå engang. Det er særlig kapittelet om design som vil være greit å ha i bakhodet gjennom denne seksjonen.

Som nevnt tidligere vil den i denne oppgaven representere en streng av bytes, nærmere bestemt meldingene vi sender og mottar mellom *master* og sensor.

Brukerenheter og ferdigstilling av profil

Etter at clusterne er ferdig definert kan de legges til en brukerenhet, eksempelvis *gateway* og HART i *figur 7.2*. Dette gjøres ved å dra clusterne med musepeker til den valgte brukerenheten. Når museknappen slippes vil det måtte bestemmes om clusteren skal være av inngående eller utgående type. Her er det viktig å gjøre riktig valg slik at binding og device discovery vil fungere for applikasjonen[11]. I *figur 7.2* er de to enhetene vist med piler mellom de clusterne som hører sammen, det vil si de med lik ID og samhørende inn/ut. Dersom en fargekopi av denne rapporten leses vil et rødt tannhjul vises foran inngående cluster, mens tannhjulet foran den utgående vil være grønn. Clusternavnene *master* og slave er benyttet for å representere HART-terminologi.



Figur 7.2: Profil for ZigBee-applikasjon.

Det som nå gjenstår for å fullføre profilen er å redigere egenskapene til brukerenhetene. Det viktigste her er å gi enhetene samme profilID, kommunikasjon vil ikke kunne foregå om enhetene tilhører forskjellige profiler. DeviceID settes til en og to for de ulike enheten, mens begge får definert et endepunkt en.

ProfileBuilder generer som sagt et sett med utgående og inngående clustere, men det ville selvsagt også være mulig å definere både ut og inn på samme cluster. Programmet må da skrives om slik at samme cluster-ID benyttes både ved sending og mottak, dette er vist i et eksempelprogram medfølgende *stack-en*. Programmets formål er å overføre seriedata trådløst for å erstatte kabling, noe lignende dette prosjektet. Dette programmet kunne selvsagt blitt benyttet og bygd ut til å omfatte den funksjonaliteten HART-modemet krever, men av årsakene nevnt innledningsvis ble det bestemt å heller bygge opp et program fra bunnen med profil.

Videreutvikling av kildekode generert av ProfileBuilder

Som det står i oppstarstsguiden[2] til ProfileBuilder generer den kun et rammeverk applikasjonsdesigneren kan benytte til videre utvikling. Det første skrittet etter generering av kildekode blir da å fjerne alle data og strukturer som ikke trengs i applikasjonen. Deretter kan vi legge til funksjonaliteten som trengs for at noden skal utføre de oppgavene vi har spesifisert. Det vil i denne seksjonen

[11] Se kapittelet for ZigBee for mer informasjon

ikke taes med alle detaljer i programmet, men heller vektlegge nøkkelfunksjoner og problemer vi kan støte på i *stacken* som benyttes. Hver oppmerksom på at delseksjoner også kan komme inn på andre temaer enn overskriften tilsier, mange programmeringsmessige temaer flyter her over i hverandre.

MT Serial - En modul for seriekommunikasjon

Stacken har en egen modul, MT Serial, som tar seg av kommunikasjon mot serieporten. Initialiseringen av seriemodulen må foregå samtidig med initialiseringen av applikasjonen den skal brukes i. Initialiseringen omfatter registrering av applikasjonen for serieporten, baudrate, minimums *gap* og flytkontroll. De to sistnevnte vil beskrives nærmere gjennom seksjonen.

En av mulighetene denne modulen gir er mottak av sammenhengende data på serieporten helt til det inntreffer et opphold på lenger enn en spesifisert tidsglipe eller *gap*. Når dette skjer vil *stackens* operativsystem, OSAL[12], gi et *event* til applikasjonen registrert for serieoppgaven. Serieapplikasjonen vil deretter behandle meldingen og eventuelt sende den videre til en annen node. Denne funksjonaliteten er valgt til å benyttes i både *gateway*-enhet og HART-enhet. Det bør nevnes at stakversjonen brukt i dette prosjektet inneholdt feil i denne modulen, noe som forhindret kompilering når seriekall ble benyttet. Problemet har forøvrig en enkel løsning:

- Lokaliser filen: MTEL.c
- Finn funksjon: void MTProcessAppRspMsg(byte *pData, byte len)
- Sett inn følgende i forkant: #if defined (ZTOOL_PORT)
- Sett inn denne i etterkant : #endif

Chipcon er nå klar over denne mangelen og ved et søk i FAQ på deres supportsider vil beskrivelsen ovenfor kunne finnes.

Nødvendig konfigurasjon

En annen ting som må huskes på når serieporten skal benyttes er å sette flagget for seriekommunikasjon i makefila. Her kan også annen funksjonalitet velges. Årsaken til denne flaggsettingen er at det muliggjør nedskalering av *stacken* når ulik funksjonalitet ikke er i bruk og dermed minske den endelige programstørrelsen.

Siden HART-modemet krever flytkontroll ble det nødvendig å sette funksjonsparameterne for dette i ZigBee-enheten som skal kommunisere direkte med modemet. Det kan forøvrig virke som om utviklerne av seriemodulen ikke har trodd at så mange vil benytte seg av denne funksjonaliteten. Parametrene som

[12] *Stacken* er bygd opp med sitt eget lille operativsystem som tar seg av scheduling av oppgaver, behandling av eventer, softwaretimere med mer. Hver oppgave må registreres i OSAL før den kan kjøre.

skal benyttes ved initialisering av flytkontrollen er ikke dokumentert noe sted, og ikke all funksjonalitet er synlig for brukeren. Under testing av ulike parameterne er det også et problem at ulike datamaskiner kan gi forskjellige resultater. Selv med lik konfigurasjon gav to av fire datamaskiner ulike resultater med terminalvindu og ZigBee-enhet innstilt på flytkontroll. Det råder fortsatt usikkerhet rundt denne problemstillingen, men en mulig årsak kan være forskjellige drivere og maskinvare på de ulike datamaskinene. Initialiseringsparameterne som til slutt ble benyttet er vist i kildekoden for enhetene vedlagt under *vedlegg C*.

En fullstendig liste over nødvendig konfigurasjon av serieporten ble gitt i steg 1 under seksjonen om driveren i LabVIEW. Med et unntak er alle disse parameterne like som *stackens*, eller kan settes med funksjoner gitt i seriemodulen. Den eneste forskjellen er at standard paritetssetting for *stacken* er satt til ingen, mens vi ønsker odde paritet. Dette er det ingen initialiseringsfunksjoner som tar seg av og det blir nødvendig å gjøre endringer i *stackmodulen* som håndterer serieoppsettet:

- Lokaliser filen: OnBoard.h
- Sett inn linjen: #define ODD_PARITY 0x30
- Lokaliser filen: OnBoard.c
- Finn funksjonen: byte SerialPortInit(void)
- Skift ut NO_PARITY med ODD_PARITY

I tillegg til paritet og flytkontroll krever også modemet driftspenning over DTR linjen, dette ble behandlet i forrige kapittelet.

Flytkontroll

Flytkontroll innebærer styring av signalene RTS (Request To Send) og CTS (Clear To Send). For vanlig kommunikasjon over RS232 defineres det enheter av typen DTE (Data Terminal Equipment), for eksempel en datamaskin, og enheter kalt DCE (Data Communications Equipment) som omfatter modem, printere og andre slave-enheter. Kommunikasjon foregår ved at en DTE-enhet setter RTS høy når den har noe å sende. Deretter venter den på at CTS skal settes høy av DCE-enheten for å indikere "klar for mottak". DTE-enheten sender så data så lenge CTS er høy. DCE-enheten kan til enhver tid sette CTS lav for å si ifra om fullt buffer og andre hendelser som krever stopp. Ved fullført overføring setter DTE-enheten RTS lav igjen. Det er nå mulig for DCE-enheten å sende data.

Det ble nevnt i forrige kapittel hvorfor det trengtes krysset kabel mellom ZigBee-enheten og modemet. For å oppsummere var årsaken til dette at vi her har to kommuniserende DCE-enheter. Det skulle imidlertid vise seg at det trengtes mer enn en krysset kabel for å få igang kommunikasjonen. I databladet til modemet kunne det leses at RTS linjen fungerte på tradisjonell måte, mens CTS linjen var koblet sammen med RTS internt i modemet. Dersom modemet kobles sammen

med en vanlig DTE-enhet vil dette sikre kommunikasjon da DTE vil motta CTS hver gang den setter RTS. Mot ZigBee-enheten oppstår det derimot et problem da den starter opp med CTS linjen høy. Dette vil sette modemet klart til å motta data samtidig som den interne sammenkoblingen av RTS og CTS vil gi ZigBee-enheten høyt signal på RTS-inngangen. Hovedproblemet ved kommunikasjonen var allikevel at ZigBee-enheten aldri satte CTS-lav etter utsending av data. Ved oscilloskop ble det påvist at data ble sendt til modemet, men aldri mottatt.

Timing og manuell styring av RTS/CTS

For å løse kommunikasjonsproblemene diskutert i forrige avsnitt var det nødvendig med full kontroll over RTS og CTS linjene. Denne funksjonaliteten var imidlertid ikke mulig å finne eller endre på da den ikke var tilgjengelig i *stacken*, men skjult i bibliotekfiler. Løsningen ble da å gå bort fra MT_serial og heller bruke funksjonalitet på et lavere abstraksjonsnivå. Grunnleggende uartfunksjonalitet var tilgjengelig gjennom *stackens hardware abstraction library* hvor det også er mulig å få tilgang til andre hardware-funksjoner både hos mikrokontroller og radio. Blant annet ble også funksjonene for å sette RTS og CTS funnet her. Når disse skal benyttes er det viktig å huske på at vi programmerer en DCE-enhet som skal oppføre seg som en DTE. RTS er definert som inngang, mens CTS er definert som utgang. Den kryssede kabelen fører til at CTS utgangen til enheten vil styre RTS-inngangen til modemet. RTS-inngangen til ZigBee-enheten trenger ikke benyttes da den allikevel ikke har noen funksjon i kommunikasjonen med modemet. Det bør også legges merke til at CTS-utgangen på ZigBee-enheten settes høy med CLR_CTS(), mens den settes lav med SET_CTS(). Altså omvendt av hva som virker naturlig når noe skal settes høyt og lavt.

Selve timingen av kommunikasjon mot modemet innebærer å sette RTS høy i så god tid at modemet rekker å klargjøre seg, samt å vente med å sette RTS lav igjen før vi er sikre på at modemet har mottatt hele meldingen. Timere basert på *busy waiting*[13] ble benyttet for å styre ventetid mellom CTS og uart-sending. Det er generelt ingen god idet å bruke slike timere når systemet inneholder et operativsystem. Grunnen til dette er at timerne kan oppta tid som kunne vært benyttet på andre oppgaver. Når disse allikevel er blitt benyttet er det for å minske kompleksiteten i systemet, ved flere og mer krevende applikasjoner ville heller software-timerne til OSAL blitt brukt.

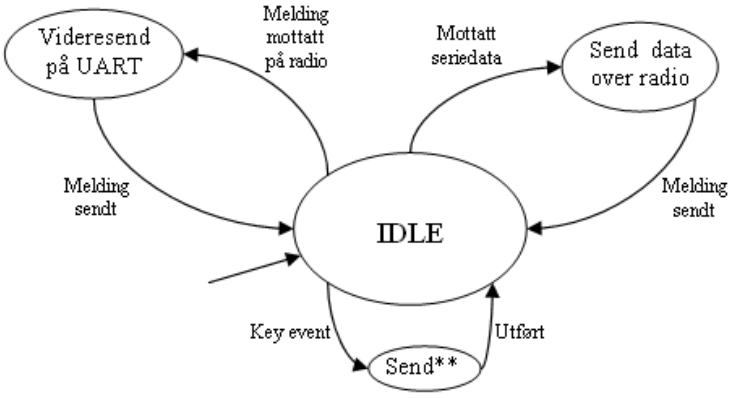
Endelig program

Tilstandsdiagrammer for de endelige programversjonene er vist i *figur 7.3* og *figur 7.4*. Valget falt naturlig på denne presentasjonsformen som følge av programmenes hendelsesbaserte natur. *Stackens* operativsystem sier ifra til applikasjonene når et *event*[14] inntreffer, denne går da fra idle-tilstand (hviletilstand) til den tilstanden som er definert for hendelsen. Enkel stjerne i tilstand betyr

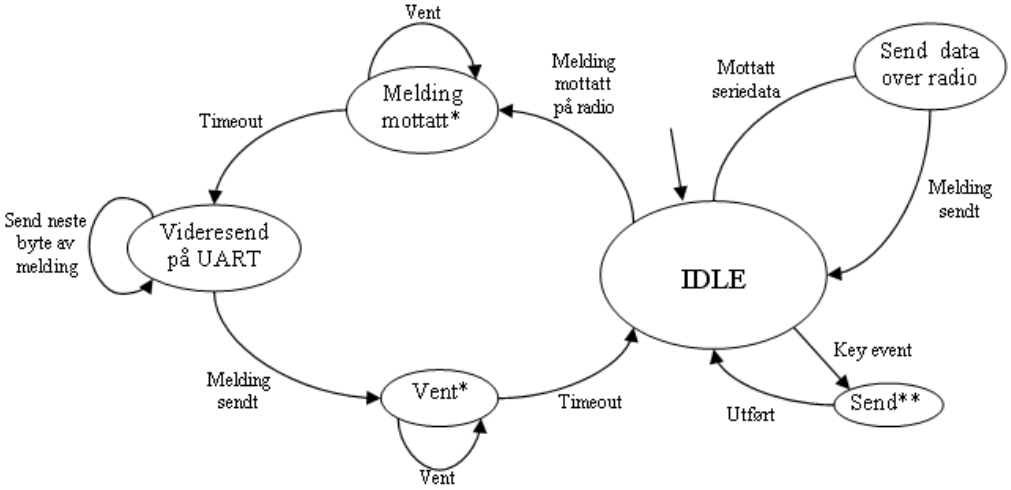
[13] Det vil si CPU konsumerende timere som går i løkke til de er ferdige.

[14] *Event* er egentlig bare det engelske ordet for hendelse. Begge ordene er brukt da *event* ofte defineres som et avbrudd som følge av en hendelse utenfor systemet.

endring av verdi på CTS-utgangen. Dobbel stjerne betyr at denne transisjonen gjelder for alle fire *key events*; *Binding Request*, *Device Discovery*, sende testmelding på radio og sende testmelding til modem. Ekstra funksjonalitet ved for eksempel sending av seriemelding er ikke vist.



Figur 7.3: Tilstandsmaskin for program i *gateway*-node.



Figur 7.4: Tilstandsmaskin for program i HART-node.

7.3 Gratis software

I løpet av prosjektet ble det eksperimentert med flere gratis program til bruk for ulike formål. Disse kan fritt lastes ned og benyttes av alle som måtte ønske. Det vil her bli presentert en liten oversikt over disse med en forklaring om hva de kan og har blitt benyttet til i løpet av prosjektet.

Det ble prøvd ut tre ulike kommunikasjonsprogram i tillegg til driveren for labVIEW. Disse var nærmere bestemt; *HART TOOL* - som kan nedlastet fra Triangle Micro Solutions[20], *Prim Ware* funnet gjennom Romilly Bowden's *software*side[21] og en avansert terminal fra Bray++[22]. Sistnevnte er egentlig ikke optimalisert for HART, men er har egenskaper som gjør den godt egnet for å sende selvgenererte meldinger. Til sist i denne seksjonen vil det også bli sett litt på et program for analyse av signaler og meldinger på serieporten. Programmet heter *Advanced Serial Port Monitor* utviklet av AGG Software[23] og er et program som kommer godt med under feilsøking av seriekommunikasjon.

HART TOOL

HART TOOL var et av de første programmene som ble forsøkt benyttet ved kommunikasjon mot sensoren. På grunn av feilen med preamble[15] ble det umulig å hente inn adresse ved polling, videre testing ble heller gjort med driveren for labVIEW som følge av egenskaper som åpen kildekode. *HART TOOL* har et greit grensesnitt og har mulighet til å innhente endel informasjon om sensoren.

Prim Ware

Prim Ware er et annet interessant program for innhenting av HART-data. Det er opprinnelig laget for dos, men så ut til å virke greit under Windows XP da dette ble forsøkt. Av samme årsak som i avsnittet over kunne heller ikke utførlig testing bli foretatt her da ingen adresse ble funnet ved søk. Det kan forøvrig leses i medfølgende dokumentasjon at *Prim Ware* inneholder funksjonalitet for skriving av de fleste universelle parameterne for HART-instrumenter. I tillegg lager den en database over feltinstrumenter og inneholder muligheter for opprettelse av brukerkommandoer. Den mest attraktive egenskapen ved *Prim Ware* er allikevel den medfølgende åpne kildekoden for programmet. Det eneste som trengs for å skreddersy sin egen applikasjon er kunnskap om c-programmering og en c-kompilator som støtter de samme bibliotekene som koden er spesifisert for. Ved å studere koden kan det også læres mye om meldingsoppbygning til HART, blant annet har den noen gode tabeller for error-respons og brukerkommandoer.

Terminal v1.9b

Terminalen fra Bray++ inneholder egenskaper som gjør den overlegen de fleste andre testprogrammer for seriekommunikasjon. Alle parametere som baudrate, databits, paritet og lignende kan enkelt endres og manipuleres mens programmet er koblet opp mot com-porten. I tillegg til vanlig strengsending har den også

[15] Preamblefeilen ble diskutert i seksjonen 'Driver for HART'

muligheter for oversending av meldinger med macroer. Det kan velges om disse skal sendes ved gitte tidsintervall eller kun når bruker velger å sende meldingen. For å illustrere hvordan en slik macrooppbygning vil se ut er vist under hvordan meldingen fra *tabell 7.1* vil være bygd opp:

- FFFFFFFF$82$26$06$21$77$86$03$00$71

Dollartegnene forteller her terminalen at dette hexadecimale tall. Det siste elementet, nærmere bestemt sjekksummen kunne nå vært skiftet ut med macroen %XOR. Sjekksummen ville da blitt regnet ut for oss og tilsvart 0x71. For mer informasjon om macroer henvises det til Brays hjemmeside[22].

Advanced Serial Port Monitor

Dette programmet kan operer i tre forskjellige modi; manuell, automatisk og spionasjemodus. Den første modien gir samme funksjonalitet som hyperterminalen medfølgende windows og er ikke noe mer spennende enn at det kan sendes og mottas vanlige tekststrenger. Atomatisk modus åpner for noe lignende funksjonalitet som terminalen fra Bray. Forskjellen fra manuell modi er at tekststrenger her kan sendes ved gitte tidsintervall. Til sist kommer den modien som er mest interessant og som har blitt benyttet ved feilsøking i dette prosjektet, nærmere bestemt spionasjemodus. Denne lar deg velge en com-port på datamaskinen og lytter deretter til all kommunikasjon som foregår her. Det er ikke bare meldingene som blir vist, men også kontrollsignal som endrer verdi. Programmet kan dessverre ikke avlytte inngående kontrollsignal, men det kan være god nok hjelp i å se den utgående kontrollen i sammenheng med meldingene. Ennå en positiv side ved programmet er brukerstøtten, ved et tilfelle ble en forespørsel besvart etter kun 5 minutters ventetid.

Del III

Avsluttende arbeid

Kapittel 8

Resultater

Resultatene for prosjektet består av den endelige ytelsen og egenskapene og til det ferdige systemet. Det vil først bli sett på optimalisering av timingverdier og ut i fra dette hvor mye *overhead* den trådløse forbindelsen har tilført kommunikasjonen sammenlignet med vanlig trådbundet overføring. Deretter vil det gis en kort beskrivelse av oppnådde egenskaper for systemet.

8.1 Optimalisering

Ved endelig testoppsett og virkende system gjenstod det å optimalisere overføringene. Dette innebar å sette gaps og timing så nær de kritiske grensene som mulig. Hver parameter ble satt så lavt at det til slutt ble oppdaget ufullstendige meldinger på skjermen. Denne ble da oppjustert og testet utførlig for å forsikre et stabilt system.

Etter finjusteringer av alle parametere ble *overhead* beregnet til å være i overkant av 20ms med både sending og mottak inkludert. Det er da kun tatt med *overhead* som følge av innlesning av seriedata. Tiden benyttet av OS-funksjoner og nettaksess vil komme i tillegg, men er anntatt å være en del lavere enn førstnevnte. Det er også anntatt at vi kun har en kjørende applikasjon i noden og at nettverket kun består av et minimum av enheter. Dersom vi har flere noder og ønsker determinisme i form av *beacons* og garanterte tidsluker må minst 15,36ms legges til i sammenlagt *overhead*.

8.2 Oppnådde egenskaper

Den ferdige trådløse *gateway*-løsningen vil enkelt kunne kobles inn istedenfor et allerede eksisterende trådbundet system. Det eneste som behøver å gjøres er å koble *gateway*-noden til datamaskinen via RS232 og koble HART-node med modem til nettverk eller sensor kompatibel for HART. Etter dette vil systemet oppføre seg akkurat som det gjorde da det var trådbundet. Det er selvsagt et kriterie at systemet takler den ekstra *overheaden* nodene medfører.

Da ZigBee-nodene nå tar seg av kontrollen for modemet trenger ikke lenger programmene i *master* å inneholde denne funksjonaliteten. Det rekommanderes å

fjerne dette da timing både her og i nodene vil føre til ekstra *overhead*. Forflytningen av kontrollfunksjonene tillater nå også bruk av andre program for utsending og mottak av meldinger. For eksempel kan terminalen fra Bray[1] nå fint utføre disse oppgavene. Det vil da være nødvendig å generere egne meldinger og det vil kreve at bruker klarer å tolke mottatte data. Disse kunnskapene kan tilegnes ved nærmere studie av kapittelet om driver for HART. For mer informasjon enn dette henvises det til HART *specification* som kan skaffes gjennom HCF[6].

I tillegg til vanlig innhenting av HART-data åpner også trådløse overføringer nye muligheter for et system. Disse vil bli diskutert nærmere i neste kapittel.

[1] Diskutert i kapittelet for gratis software.

Kapittel 9

Diskusjon

I dette kapittelet vil de ulike resultatene og egenskapene for det ferdige systemet bli diskutert. Det vil blant annet bli sett på hva multidrop-koblede HART-nettverk og flere noder i ZigBee-nettverket vil ha å si for ytelsen til innsamlingssystemet. En mulig løsning på problemstillinger som oppstår her vil bli gitt. Til sist vil det bli sett på mulighetene for en ZigBee-HART *gateway* i industrielle sammenhenger.

9.1 Optimalisert for minimum *overhead*?

I forrige kapittel ble det nevnt at innføringen av den trådløse kommunikasjonen i dette prosjektet førte til økt *overhead* i forhold til trådbundet overføring. Denne kan imidlertid reduseres ved noen kjappe inngrep. Ved å benytte funksjoner på lavere abstraksjonsnivå istedenfor avanserte høynivåfunksjoner kan vi øke ytelsen til systemet. For eksempel vil vi få et hurtigere system ved å foreta innlesninger av seriedata med grunnleggende uart-funksjoner, sammenlignet med seriemodulen i *stacken*. Grunnen til dette er at vi slipper unødvendig ventetid på gaps eller tidsgliper i innlesningen, men ekstra funksjonalitet må implementeres for å bestemme når en fullstendig melding er mottatt og klar for videresending. Som vi straks skal se vil det også være andre kilder til *overhead*.

9.2 Flernodede system med HART i multidrop

Som følge av CSMA/CA algoritmen ZigBee benytter ved nettaksess kan vi ikke garantere determinisme i et flernodet nettverk konfigurert uten *beacons*. Sjansene for kollisjoner vil øke jo fler noder som deltar i nettverket og i ekstreme situasjoner kan noder bli nektet tilgang så lenge at data ikke lenger er gyldig[1]. Dette gjelder særlig i prosessinstrumentering hvor data skal benyttes for regulering.

Dersom vi ser for oss innhentning av data fra HART-sløyfer oppkoblet i multidrop vil vi kunne redusere *overhead* ved å samle opp data fra ulike pollinger og oversende dem i samme melding. Dette vil redusere sjansen for kollisjon med

[1] Dersom en node blir uendelig nedprioritert til fordel for andre kalles det utsultning, men det blir ikke helt riktig å benytte dette uttrykket her.

andre noder da vi ikke får så hyppige oversendinger. Dersom *beacons* benyttes og vi vil fordele de syv mulige garanterte tidslukene på ulike noder vil det være hensiktsmessig å vite hvor mange bytes som kan oversendes i hver tidsluke. Det er forsøkt å gjøre beregninger på dette, men det er ikke funnet noen informasjon i spesifikasjonene angående emnet.

Om vi har et innsamlingssystem som også henter inn data fra andre sensorer og nettverk kan det tenkes å benytte samme metodikk som ovenfor for å begrense *overhead*. Nemlig ved å la noen noder hente inn data fra flere andre og sende denne informasjonen videre i større meldinger.

9.3 Muligheter for en ZigBee-HART *gateway*

Som det allerede er nevnt åpner trådløs kommunikasjon for mange nye muligheter i instrumenteringsverdenen. Det vil blant annet bli mye enklere å hente ut data fra sensorer plassert på roterende gjenstander. Løsninger som tidligere har vært basert på sleperinger kan nå skiftes ut med påmonterte trådløse noder og vi slipper problemer som følge av slitasje og midlertidige feil. Sensorer plassert i andre vanskelig kablede miljøer kan også tenkes og installasjonsarbeidet vil være betraktelig forenklet.

Det vil selvsagt også være situasjoner hvor trådløse overføringer vil være utilstrekkelig. Ofte kreves høyere hastigheter og strengere deterministiske krav enn trådløs kommunikasjon kan gi. Topologier som mesh tilbudt av ZigBee gjør allikevel at trådløse nettverk blir tatt mer på alvor i stadig flere oppgaver.

Batteriforbruk er et annet område som bør tas med i betraktningen når et trådløst nettverk vurderes. I dette prosjektet kunne det for eksempel vært vurdert å hente driftspenning fra 4-20mA sløyfen.

Del IV

Vedlegg og bibliografi

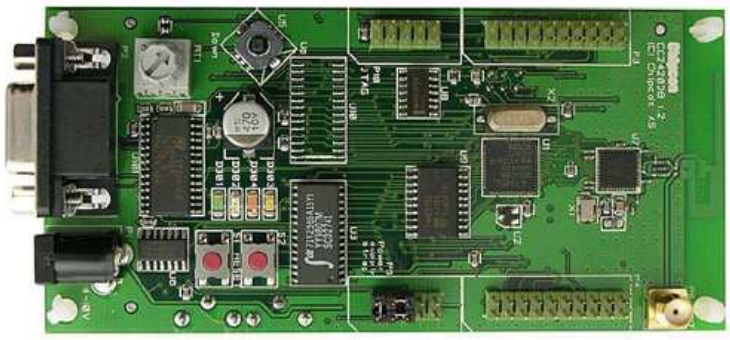
Tillegg A

Utstyr og enheter

I dette vedlegget vil det ulike utstyret og enhetene brukt i prosjektet bli vist. Det vil også bli gitt en kortfattet forklaring på bruk av nodene og hvordan de skal initialiseres for å kunne snakke sammen.

A.1 CC2420DB - ZigBee utviklingskort

I *figur A.1* er Chipcons utviklingskort CC2420DB vist. Det som er spesielt viktig å legge merke til her er plasseringen av jumperne sentrert nederst på bildet. Uten disse jumperinnstillingene vil ikke de utviklede applikasjonene virke som de skal. Jumperen til venstre aktiverer spenningstilførselen til kretskortet. Om ønskelig kan det kobles inn et multimeter her for å sjekke strømforbruk. Jumperen nederst til høyre medfører spenning til temperatursensoren og potensiometeret på kortet, denne er ikke nødvendig for vår applikasjon. Den siste jumperen er for å tvinge RS232 driveren til alltid å være på. Dette er vesentlig for at den utviklede programvaren skal fungere. I brukermanualen kan det leses at jumperen kan flyttes til en annen posisjon for å gi software-kontroll av driveren. Det er viktig å vite at software-kontroll ikke er det samme som flytkontroll, men styring av spenning til selve RS232-kretsen. Legg også merke til *joysticken* øverst til venstre da denne kommer til å bli brukt i seksjonen 'oppstart og bruk av utstyr'.

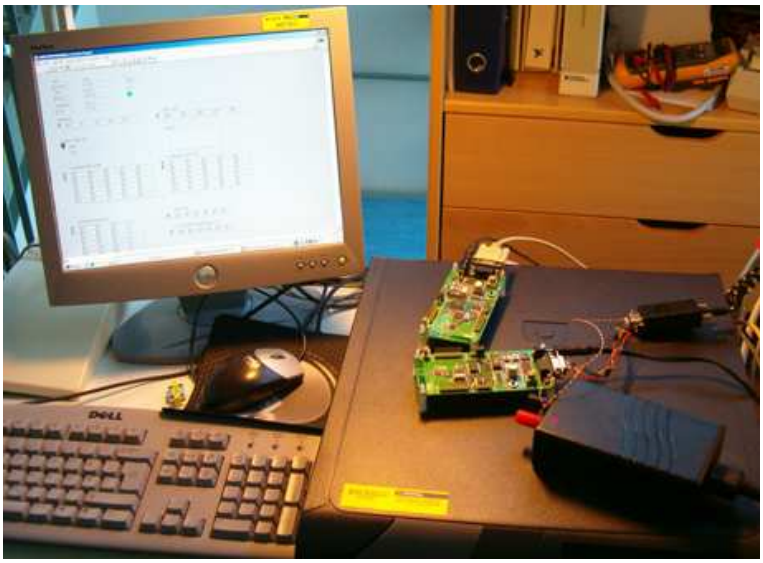


Figur A.1: CC2420DB - utviklingskort fra Chipcon.

A.2 Testoppsett

Denne seksjonen viser bilder av det endelige testoppsettet. Dessverre er testriggen snudd motsatt vei av der utstyret er montert, så vi må nøye oss med koblingsracket på baksiden. I *figur A.3* er imidlertid tilkoblingen av modem til trykksensoren vist. *Figur A.2* og *figur A.3* viser testoppsettet i ulike vinkler.

Figur A.2: Fullstendig testoppsett på Rotvoll.



Figur A.3: Testoppsett vist uten demorigg.

Figur A.4: Tilkobling av modem til trykksensor.

A.3 Oppstart og bruk av utstyr

Første gang nodene skal taes i bruk må utviklingskortene programmeres. Ferdigkompilerte hex-filer ligger vedlagt på CD, men om det er ønskelig å gjøre endringer i applikasjonen er en kort forklaring gitt her. For å slippe å gjøre endringer i makefila kan kildefilene plasseres i tilsvarende mappe de har ligget i på utviklingsmaskinen. Kopier mappene HART og *Gateway* fra vedlagt cd til mappen;

```
C:\Chipcon\CC-1.0-1.1.0\Z-Stack\Projects\Samples
```

Prosjektene kan nå åpnes i programmers notepad og kompileres for opplastning til utviklingskortene med JTAG eller ISP.

Etter at nodene er programmert med riktige program må nodene initialiseres. *Gateway*-noden startes opp først da denne er koordinator og ansvarlig for å opprettelse av nettverket. Når HART-noden så startes opp vil den søke etter koordinatoren og få tildet en kort adresse. Når dette er gjort vil den røde dioden på kortene lyse og vi har dannet et nettverk.

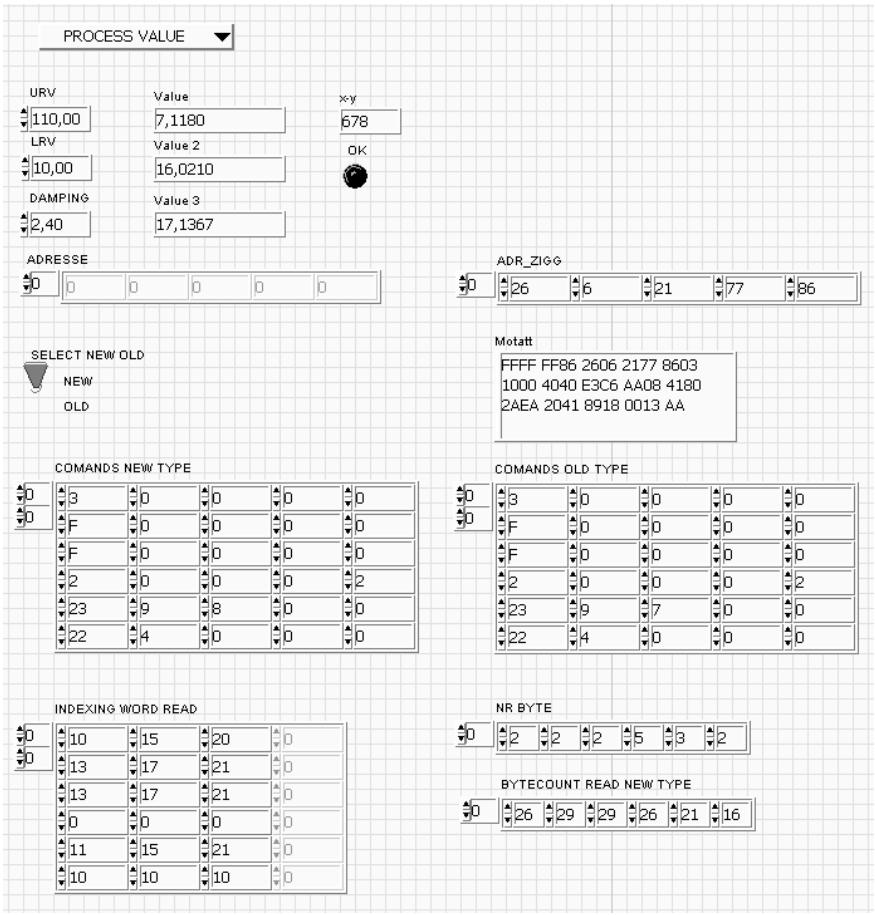
For å gjøre tjenestene til nodene tilgjengelige for hverandre kan enten device discovery eller binding benyttes. Ved binding trykkes *joysticken* på HART-noden mot høyre, inn mot midten av kortet. For å besvare forespørselen må det samme gjøres på *gateway* innen noen få sekunder. Den vil da sende en *accept* på *binding requesten* om denne var vellykket. Om nodene har kontakt med hverandre vil den gule dioden på de to nå lyse. Det er isåfall bare å koble dem opp og sende meldinger fra og til en sensor. *Device Discovery* foretas ved å trykke joystick mot venstre, bort fra kortet. Denne prosessen må også gjentas for begge noder.

I HART-noden er det også mulig å sende testmeldinger for å sjekke at kommunikasjonen virker. Ved å trykke *joystick* opp vil det sendes en melding ut på serieporten, nærmere bestemt meldingen for polling etter prosessverdi. Trykkes joystick nedover sendes en testmeldingen ut på radioen.

Tillegg B

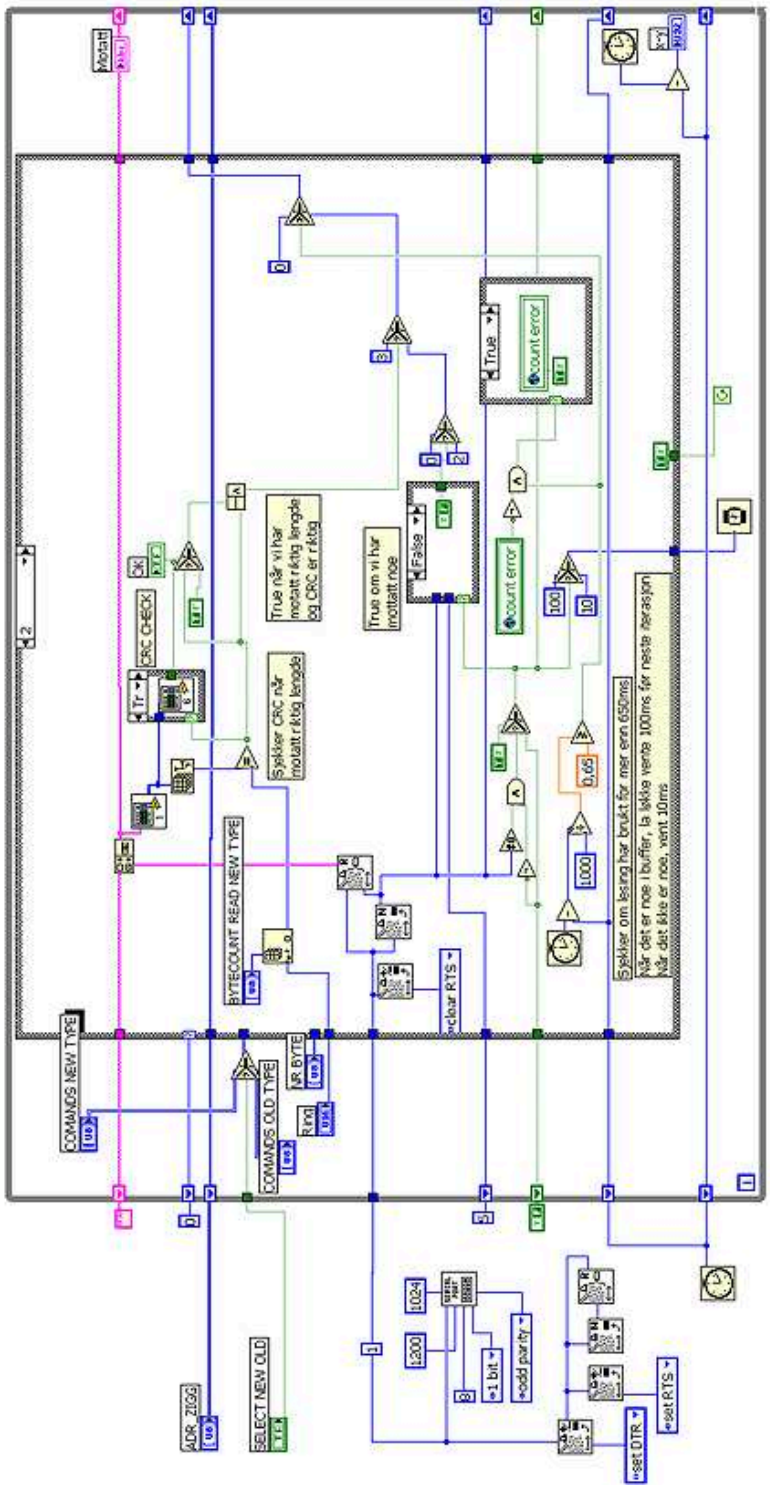
Driver for labVIEW

B.1 Brukergrensesnitt



Figur B.1: Driver for HART i labVIEW - brukergrensesnitt.

B.2 Kodeblokker



Figur B.2: Driver for HART i labVIEW - Kodeblokker.

Tillegg C

Kildekode for ZigBee

Kildekoden for begge nodene består av tre filer; applikasjonsfil, headerfil for applikasjonfilen og en fil for å registrere oppgaver i operativsystemet, OSAL. Av programmene for de to nodene inneholder HART-noden mest funksjonalitet, det er her også lagt til fler forklarende kommentarer.

C.1 Kildekode for HART-node

HART.c

```
/*********************************************************************

  ZigBee Node For HART Communication

  by Nils Petter Eftedal

  Autumn 2005


  Copyright (c) 2005 by Figure 8 Wireless, Inc., All Rights Reserved.
  Permission to use, reproduce, copy, prepare derivative works,
  modify, distribute, perform, display or sell this software and/or
  its documentation for any purpose is prohibited without the express
  written consent of Figure 8 Wireless, Inc.*/

/*********************************************************************
 * INCLUDES
 */
#include "OSAL.h"
#include "AF.h"
#include "ZDApp.h"
#include "MTSPCI.h"

#include "HART.h"
#include "OnBoard.h"
#include <avr/delay.h>


/*********************************************************************
 * CONSTANTS
 */
#define NUM_ENDPOINTS 1
#define STARTING_ENDPOINT 1

/* For the delay loop defines 8MHz Clock */
#define F_CPU 8000000UL

/*********************************************************************
 * GLOBAL VARIABLES
 */

// Task ID for internal task/event processing
byte HART_TaskID;
// This variable will be received when HART_Init() is called.
devStates_t HART_NwkState;
// This is the unique message ID (counter)
byte HART_TransID;
```

```
// Destination addresses
afAddrType_t HART_DstAddr[NUM_ENDPOINTS];

// This is the Cluster ID List
const byte HART_InClusterList[HART_MAX_IN_CLUSTERS] =
{
  HART_MASTER_CLUSTER_ID
};

const byte HART_OutClusterList[HART_MAX_OUT_CLUSTERS] =
{
  HART_SLAVE_CLUSTER_ID
};

const SimpleDescriptionFormat_t HART_SimpleDesc[NUM_ENDPOINTS] =
{
  {
    HART_ENDPOINT_1,              //  int     Endpoint;
    HART_PROFID,                  //  uint16  AppProfId[2];
    HART_DEVICEID,                //  uint16  AppDeviceId[2];
    HART_DEVICE_VERSION,          //  int     AppDevVer:4;
    HART_FLAGS,                   //  int     AppFlags:4;
    HART_MAX_IN_CLUSTERS,         //  byte    AppNumInClusters;
    (byte*)HART_InClusterList,    //  byte   *pAppInClusterList;
    HART_MAX_OUT_CLUSTERS,        //  byte    AppNumOutClusters;
    (byte*)HART_OutClusterList    //  byte   *pAppOutClusterList;
  },

};

// This is the Endpoint description.
const endPointDesc_t HART_epDesc[NUM_ENDPOINTS] =
{
  {
    HART_ENDPOINT_1,
    &HART_TaskID,
    (SimpleDescriptionFormat_t *)&HART_SimpleDesc[0],
    noLatencyReqs
  },

};


/*********************************************************************
 * LOCAL FUNCTIONS
 */
void MessageKVPCB( afIncomingKVPPacket_t *pckt );
void MessageMSGCB( afIncomingMSGPacket_t *pckt );
byte IndexFromEndpoint( byte endpoint );
void HandleKeys( byte shift, byte keys );
void HART_SendMessage_SLAVE( byte *msg);
void HARTEN_TestSend( void );
void sendUART(char Data[], int antall);

/*********************************************************************
 * PUBLIC FUNCTIONS
 */

/*********************************************************************
 * @fn      HART_Init
 *
 * @brief   Initialization function for the Generic App Task.
 *          This is called during initialization and should contain
 *          any application specific initialization (ie. hardware
 *          initialization/setup, table initialization, power up
 *          notificaiton ... ).
 *
 * @param   task_id - the ID assigned by OSAL.  This ID should be
 *                    used to send messages and set timers.
 *
 * @return  none
 */
void HART_Init( byte task_id )
{
  byte x;

  HART_TaskID   = task_id;
  HART_NwkState = DEV_INIT;
  HART_TransID  = 0;

  // Device hardware initialization can be added here or in main() (Zmain.c).

  // Register the endpoint description with the AF
  for ( x = 0; x < NUM_ENDPOINTS; x++ )
  {
    HART_DstAddr[x].addrMode = AddrNotPresent;
    HART_DstAddr[x].endPoint = 0;
    HART_DstAddr[x].addr.shortAddr = 0;
    afRegister( (endPointDesc_t *)&(HART_epDesc[x]) );
  }
```

```
  // Register for all key events - This app will handle all key events
  RegisterForKeys( HART_TaskID );

  // Register Task_ID with MTSPCI so serial messages know where to go
  MT_SerialRegisterTaskID( HART_TaskID );

  // Set the serial I/O baud rate
  MT_SetZAppBaudRate( BR_1200 );

  // Set the amount of time to wait for a quiet time.  This will determine
  // the end of a message/buffer. The serial driver will collect serial data
  // in its buffer, until this amount of quiet time, then it will send the
  // buffer to this application. This quiet time is in milliseconds.
  MT_SetZAppMinGap( 10 );

  // Enable App Flow Control. This allows the serial driver to send collected
  // serial data to this application. App Flow Control permits this application
  // to determine when to allow serial data to be delivered from the driver.

  MT_SerialFlowControl( ZAPP_PORT, SFC_AP_EN );


}

/*********************************************************************
 * @fn      HART_ProcessEvent
 *
 * @brief   Generic Application Task event processor.  This function
 *          is called to process all events for the task.  Events
 *          include timers, messages and any other user defined events.
 *
 * @param   task_id  - The OSAL assigned task ID.
 * @param   events   - Events to process.  This is a bit map and can
 *                     contain more than one event.
 *
 * @return  none
 */
void HART_ProcessEvent( byte task_id, UINT16 events )
{
  osal_msg_received_t   *msg;
  afIncomingKVPPacket_t *KVPpkt;
  afIncomingMSGPacket_t *MSGpkt;
  byte                  *msgPtr;
  byte                   dstEP;
  byte                   ep;
  zAddrType_t           *dstAddr;
  byte                  sentEP;
  byte                  sentStatus;
  byte                  sentTransID;
  byte                  x;

  if ( events & SYS_EVENT_MSG )
  {
    msg = osal_msg_receive( HART_TaskID );
    while ( msg )
    {
      msgPtr = msg->msg_ptr;

      switch ( *msgPtr )
      {
        case KEY_CHANGE:
          HandleKeys( msgPtr[KEY_CHANGE_SHIFT_IDX], msgPtr[KEY_CHANGE_KEYS_IDX] );
          break;

        case AF_DATA_CONFIRM_CMD:
          // This message is received as a confirmation of a data packet sent.
          // The status is of ZStatus_t type [defined in NLMEDE.h]
          // The message fields are defined in AF.h
          sentEP      = msgPtr[AF_DATA_CONFIRM_ENDPOINT];
          sentStatus  = msgPtr[AF_DATA_CONFIRM_STATUS];
          sentTransID = msgPtr[AF_DATA_CONFIRM_TRANSID];

          // Action taken when confirmation is received.
          /* Put code here */
          break;

                //Innkommende Seriemelding
        case MT_INCOMING_ZAPP_PORT:

                  /* Blink grønn diode */
                  BlinkLed( LED1, 1, 90, 90 );

          // Sender innkommende seriemelding til Gateway
                HART_SendMessage_SLAVE( msgPtr );

          break;


        case AF_INCOMING_KVP_CMD:
```

```
        case AF_INCOMING_GRP_KVP_CMD:
          // convert to incoming packet format
          KVPpkt = (afIncomingKVPPacket_t *)&(msgPtr[1]);

          // Process the incoming message
          MessageKVPCB( KVPpkt );

          // Release the data buffer
          osal_mem_free( KVPpkt->cmd.Data );
          break;

        case AF_INCOMING_MSG_CMD:
        case AF_INCOMING_GRP_MSG_CMD:
          // convert to incoming packet format
          MSGpkt = (afIncomingMSGPacket_t *)&(msgPtr[1]);

          // Process the incoming message
          MessageMSGCB( MSGpkt );

          // Release the data buffer
          osal_mem_free( MSGpkt->cmd.Data );
          break;

        case ZDO_NEW_DSTADDR:
          dstEP = msgPtr[ZDO_NEW_DSTADDR_DSTEP];
          dstAddr = (zAddrType_t *)&msgPtr[ZDO_NEW_DSTADDR_DSTADDR];
          ep = msgPtr[ZDO_NEW_DSTADDR_EP];
          // Change the endpoint into an index
          x = IndexFromEndpoint( ep );
          HART_DstAddr[x].addrMode = dstAddr->addrMode;
          HART_DstAddr[x].endPoint = dstEP;

          if ( dstAddr->addrMode == Addr16Bit )
            HART_DstAddr[x].addr.shortAddr = dstAddr->addr.shortAddr;
          else
          {
            osal_memcpy( HART_DstAddr[x].addr.extAddr,
                         dstAddr->addr.extAddr, Z_EXTADDR_LEN );
          }
          break;

        case ZDO_STATE_CHANGE:
          HART_NwkState = (devStates_t)msgPtr[1];
          if ( (HART_NwkState == DEV_ZB_COORD)
               || (HART_NwkState == DEV_ROUTER)
               || (HART_NwkState == DEV_END_DEVICE) )
          break;

        default:
          break;
      }

      // Release the memory
      osal_msg_deallocate( msgPtr );

      // Next
      msg = osal_msg_receive( HART_TaskID );
    }
  }

  // Send a message out - This event is generated by a timer
  //  (setup in HART_Init()).
  if ( events & HART_SEND_MSG_EVT )
  {
      for( x = 0; x < NUM_ENDPOINTS; x++ )
      {
        ep = HART_epDesc[x].endPoint;
      }
    }
}

/*****************************************************************
 * Event Generation Functions
 */
/*****************************************************************
 * @fn      HandleKeys
 *
 * @brief   Handles all key events for this device.
 *
 * @param   shift - true if in shift/alt.
 * @param   keys - bit field for key events. Valid entries:
 *                 EVAL_SW4
 *                 EVAL_SW3
 *                 EVAL_SW2
 *                 EVAL_SW1
 *
 * @return  none
 */
void HandleKeys( byte shift, byte keys )
{
```

```
    byte x;

  // Shift is used to make each button/switch dual purpose.
  if ( shift )
  {
    if ( keys & EVAL_SW1 )
    {
    }
    if ( keys & EVAL_SW2 )
    {
    }
    if ( keys & EVAL_SW3 )
    {
    }
    if ( keys & EVAL_SW4 )
    {
    }
  }
  else
  {
    if ( keys & EVAL_SW1 )
    {

          /* This code will send out a request for process value to the
          HART modem, using the serial port. The adress needs to be
          modified when different sensors are to be used. */

          int noe=13;
          char test[14] = {0xFF, 0xFF, 0xFF, 0xFF, 0x82, 0x26, 0x06, 0x21, 0x77, 0x86
               , 0x03, 0x00, 0x71};
          sendUART(test, noe);

          /* Blink on green diode */
          BlinkLed( LED1, 1, 90, 90 );

    }

    if ( keys & EVAL_SW2 )
    {
      // Initiate an End Device Bind Request for the mandatory endpoint
      for( x = 0; x < NUM_ENDPOINTS; x++ )
        ZDApp_SendEndDeviceBindReq( HART_epDesc[x].endPoint );
    }

    if ( keys & EVAL_SW3 )
    {
          /* This code sends out a seven character long testmessage. The
          message may be modified as you wish. Remember to change the
          length character, currently 7, to the length of the message
          if you decide to modify */

          char *testPtr;
          char test[10] = {0, 7, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77};
          test[9]='\0';
          testPtr = test;

          /* Send out an Over The Air testmessage to gateway */
          HART_SendMessage_SLAVE( testPtr );
    }

    if ( keys & EVAL_SW4 )
    {
      // Initiate a Match Description Request (Service Discovery)
      //  for the mandatory endpoint
      for( x = 0; x < NUM_ENDPOINTS; x++ )
        ZDApp_AutoFindDestination( HART_epDesc[x].endPoint );
    }
  }
}

/*********************************************************************
 * LOCAL FUNCTIONS
 */

/*********************************************************************
 * @fn      MessageKVPCB
 *
 * @brief   Data message processor callback.  This function processes
 *          any incoming data - probably from other devices.  So, based
 *          on cluster ID, perform the intended action.
 *
 * @param   none
 *
 * @return  none
 */
void MessageKVPCB( afIncomingKVPPacket_t *pkt )
{

  switch ( pkt->clusterId )
```

```
  {
  }
}

/*********************************************************************
 * @fn      MessageMSGCB
 *
 * @brief   Data message processor callback.  This function processes
 *          any incoming data - probably from other devices.  So, based
 *          on cluster ID, perform the intended action.
 *
 * @param   none
 *
 * @return  none
 */
void MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
  switch ( pkt->clusterId )
  {

    case HART_MASTER_CLUSTER_ID:

          /* Send motatt data ut på serieporten */
          //MT_SerialSendZAppMsg(pkt->cmd.Data, pkt->cmd.DataLength);

          /* Prøver min nye utsendingsmetode */
          sendUART(pkt->cmd.Data, pkt->cmd.DataLength);

      break;
  }
}

/*********************************************************************
 * @fn      HART_SendMessage_SLAVE
 *
 * @brief   Send a standard type message.
 *
 * @param   none
 *
 * @return  none
 */
void HART_SendMessage_SLAVE( byte *msg )
{
  afStatus_t stat;

  // This message is setup to use APS ACKs (application layer acknowledgements)
  // APS ACKs take up bandwidth.  If you would like to increase the transmit
  // rate and you don't care about ACKing at the application, turn off APS ACK
  // by replacing AF_MSG_ACK_REQUEST with 0 in the function call below.
  stat = afFillAndSendMessage( &HART_DstAddr[0], HART_epDesc[0].endPoint,
                               HART_SLAVE_CLUSTER_ID, 1, FRAMETYPE_MSG,
                               &HART_TransID, NULL, NULL, NULL, NULL,
                               msg[1], &msg[2],
                               AF_MSG_ACK_REQUEST, true, AF_DEFAULT_RADIUS );

  if ( stat != ZSuccess );

  else
    // Visual indication that message went out
    BlinkLed( LED1, 1, 90, 90 );
}


/*********************************************************************
 * @fn      IndexFromEndpoint
 *
 * @brief   Utility function that returns the index for the passed in endpoint.
 *
 * @param   endpoint - which endpoint
 *
 * @return  index into the endpoint descriptor
 *          NUM_ENDPOINTS if not found
 */
byte IndexFromEndpoint( byte endpoint )
{
  byte x;

  for ( x = 0; x < NUM_ENDPOINTS; x++ )
  {
    if ( endpoint == HART_epDesc[x].endPoint )
      break;
  }
  return x;
}


/* This function is for testing purposes only. If you rather would like to
send a text string than an array of individual characters this function may
be used instead of the one in "Joystick down" event. */
```

```
void HARTEN_TestSend( void )
{
  byte theMessageData[] = "Hello World";

  if ( afFillAndSendMessage( &HART_DstAddr[0], HART_epDesc[0].endPoint,
       HART_SLAVE_CLUSTER_ID,
                              1, FRAMETYPE_MSG, &HART_TransID,
                              NULL, NULL, NULL, ERRORCODE_SUCCESS,
                              (byte)osal_strlen(theMessageData), (byte*)&
                                   theMessageData,
                              AF_MSG_ACK_REQUEST, true, AF_DEFAULT_RADIUS ))
  {
    // Error occurred in request to send
  }
  else
  {
    // Successfully requested to be sent
  }
}


/* This funtion is used to send data out to the HART modem, via
the serial port. It is used instead of the MT serial to have
fully control over the serial control line RTS. Here defines as
CTS because the stack is actually made to be in slave mode to a
computer */

void sendUART(char Data[], int antall)
{
        int i, lengde;
        lengde = antall;

        /* Set RTS high */
        CLR_CTS();

        /* Give HART time to initialize */
        _delay_ms ( 10 );

    UART1_SEND(Data[0]);
    for (i = 1; i < lengde; i++) {
        UART1_WAIT_AND_SEND(Data[i]);
    }

        /* Wait to ensure that HART recieves */
        _delay_ms ( 200 );

        /* Set RTS low */
        SET_CTS();
}
```

HART.h

```
#ifndef HART_H
#define HART_H
/*********************************************************************
    Copyright (c) 2005 by Figure 8 Wireless, Inc., All Rights Reserved.
    Permission to use, reproduce, copy, prepare derivative works,
    modify, distribute, perform, display or sell this software and/or
    its documentation for any purpose is prohibited without the express
    written consent of Figure 8 Wireless, Inc.
*********************************************************************/

#ifdef __cplusplus
extern "C"
{
#endif

/*********************************************************************
 * INCLUDES
 */
#include "ZComDef.h"

/*********************************************************************
 * CONSTANTS
 */

#define HART_SEND_MSG_TIMEOUT   1000
#define HART_SEND_MSG_EVT       0x0001
#define HART_FLAGS              0

#define HART_ENDPOINT_1 1
#define HART_PROFID 2
#define HART_DEVICEID 2
#define HART_DEVICE_VERSION 1
#define HART_MAX_OUT_CLUSTERS 1
#define HART_MAX_IN_CLUSTERS 1
#define HART_MASTER_CLUSTER_ID 1
#define HART_SLAVE_CLUSTER_ID 2

/*
 * Task Initialization for the Generic Application
 */
extern void HART_Init( byte task_id );

/*
 * Task Event Processor for the Generic Application
 */
extern void HART_ProcessEvent( byte task_id, UINT16 events );

/*********************************************************************
*********************************************************************/

#ifdef __cplusplus
}
#endif

#endif /* HART_H */
```

OSAL_HART.c

```
/**************************************************************************
  Copyright (c) 2005 by Figure 8 Wireless, Inc., All Rights Reserved.
  Permission to use, reproduce, copy, prepare derivative works,
  modify, distribute, perform, display or sell this software and/or
  its documentation for any purpose is prohibited without the express
  written consent of Figure 8 Wireless, Inc.
 **************************************************************************
 **************************************************************************
 * INCLUDES
 */
#include "ZComDef.h"
#include "OSAL.h"
#include "OSAL_Tasks.h"

#if defined ( MT_TASK )
  #include "MTEL.h"
#endif

#if !defined( NONWK )
  #include "nwk.h"
  #include "APS.h"
  #include "ZDApp.h"
#endif

#include "HART.h"
#include "OnBoard.h"


/**************************************************************************
 * FUNCTIONS
 **************************************************************************/

/**************************************************************************
 * @fn      osalAddTasks
 *
 * @brief   This function adds all the tasks to the task list.
 *          This is where to add new tasks.
 *
 * @param   void
 *
 * @return  none
 */
void osalAddTasks( void )
{
#if defined( MT_TASK )
  osalTaskAdd( MT_TaskInit, MT_ProcessEvent );
#endif

  osalTaskAdd( OnBoard_TaskInit, OnBoard_ProcessEvent );

  osalTaskAdd( nwk_init, nwk_event_loop );
  osalTaskAdd( APS_Init, APS_event_loop );
  osalTaskAdd( ZDApp_Init, ZDApp_event_loop );
  osalTaskAdd( HART_Init, HART_ProcessEvent );
}
```

C.2 Kildekode for *Gateway*-node

Gateway.c

```
/**************************************************************************

  ZigBee GATEWAY NODE For HART Communication

  by Nils Petter Eftedal

  Autumn 2005


  Copyright (c) 2005 by Figure 8 Wireless, Inc., All Rights Reserved.
  Permission to use, reproduce, copy, prepare derivative works,
  modify, distribute, perform, display or sell this software and/or
  its documentation for any purpose is prohibited without the express
  written consent of Figure 8 Wireless, Inc.*/

/**************************************************************************
 * INCLUDES
 */
#include "OSAL.h"
#include "MTSPCI.h"
#include "AF.h"
#include "ZDApp.h"
```

```
#include "Gateway.h"
#include "OnBoard.h"


/*********************************************************************
 * CONSTANTS
 */
#define NUM_ENDPOINTS 1
#define STARTING_ENDPOINT 1


/*********************************************************************
 * GLOBAL VARIABLES
 */

// Task ID for internal task/event processing
byte Gateway_TaskID;
// This variable will be received when Gateway_Init() is called.
devStates_t Gateway_NwkState;
// This is the unique message ID (counter)
byte Gateway_TransID;
// Destination addresses
afAddrType_t Gateway_DstAddr[NUM_ENDPOINTS];

// This is the Cluster ID List
const byte Gateway_InClusterList[GATEWAY_MAX_IN_CLUSTERS] =
{
  GATEWAY_SLAVE_CLUSTER_ID
};

const byte Gateway_OutClusterList[GATEWAY_MAX_OUT_CLUSTERS] =
{
  GATEWAY_MASTER_CLUSTER_ID
};

const SimpleDescriptionFormat_t Gateway_SimpleDesc[NUM_ENDPOINTS] =
{
  {
    GATEWAY_ENDPOINT_1,                 //  int     Endpoint;
    GATEWAY_PROFID,                     //  uint16  AppProfId[2];
    GATEWAY_DEVICEID,                   //  uint16  AppDeviceId[2];
    GATEWAY_DEVICE_VERSION,             //  int     AppDevVer:4;
    GATEWAY_FLAGS,                      //  int     AppFlags:4;
    GATEWAY_MAX_IN_CLUSTERS,            //  byte    AppNumInClusters;
    (byte*)Gateway_InClusterList,       //  byte   *pAppInClusterList;
    GATEWAY_MAX_OUT_CLUSTERS,           //  byte    AppNumOutClusters;
    (byte*)Gateway_OutClusterList       //  byte   *pAppOutClusterList;
  },

};

// This is the Endpoint description.
const endPointDesc_t Gateway_epDesc[NUM_ENDPOINTS] =
{
  {
    GATEWAY_ENDPOINT_1,
    &Gateway_TaskID,
    (SimpleDescriptionFormat_t *)&Gateway_SimpleDesc[0],
    noLatencyReqs
  },

};


/*********************************************************************
 * LOCAL FUNCTIONS
 */
void MessageKVPCB( afIncomingKVPPacket_t *pckt );
void MessageMSGCB( afIncomingMSGPacket_t *pckt );
byte IndexFromEndpoint( byte endpoint );
void HandleKeys( byte shift, byte keys );
void Gateway_SendMessage_MASTER( byte *msg );


/*********************************************************************
 * PUBLIC FUNCTIONS
 */

/*********************************************************************
 * @fn      Gateway_Init
 *
 * @brief   Initialization function for the Generic App Task.
 *          This is called during initialization and should contain
 *          any application specific initialization (ie. hardware
 *          initialization/setup, table initialization, power up
 *          notificaiton ... ).
 *
 * @param   task_id - the ID assigned by OSAL.  This ID should be
 *                    used to send messages and set timers.
```

```
 *
 * @return  none
 */
void Gateway_Init( byte task_id )
{
  byte x;

  Gateway_TaskID   = task_id;
  Gateway_NwkState = DEV_INIT;
  Gateway_TransID  = 0;

  // Device hardware initialization can be added here or in main() (Zmain.c).

  // Register the endpoint description with the AF
  for ( x = 0; x < NUM_ENDPOINTS; x++ )
  {
    Gateway_DstAddr[x].addrMode = AddrNotPresent;
    Gateway_DstAddr[x].endPoint = 0;
    Gateway_DstAddr[x].addr.shortAddr = 0;
    afRegister( (endPointDesc_t *)&(Gateway_epDesc[x]) );
  }

  // Register for all key events - This app will handle all key events
  RegisterForKeys( Gateway_TaskID );

  // Register Task_ID with MTSPCI so serial messages know where to go
  MT_SerialRegisterTaskID( Gateway_TaskID );

  // Set the serial I/O baud rate
  MT_SetZAppBaudRate( BR_1200 );

  // Set the amount of time to wait for a quiet time.  This will determine
  // the end of a message/buffer. The serial driver will collect serial data
  // in its buffer, until this amount of quiet time, then it will send the
  // buffer to this application. This quiet time is in milliseconds.
  MT_SetZAppMinGap( 10 );

  // Enable App Flow Control. This allows the serial driver to send collected
  // serial data to this application. App Flow Control permits this application
  // to determine when to allow serial data to be delivered from the driver.
  MT_SerialFlowControl( ZAPP_PORT, SFC_AP_EN );

}

/*********************************************************************
 * @fn      Gateway_ProcessEvent
 *
 * @brief   Generic Application Task event processor.  This function
 *          is called to process all events for the task.  Events
 *          include timers, messages and any other user defined events.
 *
 * @param   task_id  - The OSAL assigned task ID.
 * @param   events   - Events to process.  This is a bit map and can
 *                     contain more than one event.
 *
 * @return  none
 */
void Gateway_ProcessEvent( byte task_id, UINT16 events )
{
  osal_msg_received_t    *msg;
  afIncomingKVPPacket_t *KVPpkt;
  afIncomingMSGPacket_t *MSGpkt;
  byte                   *msgPtr;
  byte                    dstEP;
  byte                    ep;
  zAddrType_t            *dstAddr;
  byte                   sentEP;
  byte                   sentStatus;
  byte                   sentTransID;
  byte                   x;

  if ( events & SYS_EVENT_MSG )
  {
    msg = osal_msg_receive( Gateway_TaskID );
    while ( msg )
    {
      msgPtr = msg->msg_ptr;

      switch ( *msgPtr )
      {
        case KEY_CHANGE:
          HandleKeys( msgPtr[KEY_CHANGE_SHIFT_IDX], msgPtr[KEY_CHANGE_KEYS_IDX] );
          break;

        case AF_DATA_CONFIRM_CMD:
          // This message is received as a confirmation of a data packet sent.
          // The status is of ZStatus_t type [defined in NLMEDE.h]
          // The message fields are defined in AF.h
          sentEP      = msgPtr[AF_DATA_CONFIRM_ENDPOINT];
          sentStatus  = msgPtr[AF_DATA_CONFIRM_STATUS];
```

```
            sentTransID = msgPtr[AF_DATA_CONFIRM_TRANSID];

            // Action taken when confirmation is received.
            /* Put code here */
            break;

          case AF_INCOMING_KVP_CMD:
          case AF_INCOMING_GRP_KVP_CMD:
            // convert to incoming packet format
            KVPpkt = (afIncomingKVPPacket_t *)&(msgPtr[1]);

            // Process the incoming message
            MessageKVPCB( KVPpkt );

            // Release the data buffer
            osal_mem_free( KVPpkt->cmd.Data );
            break;

                  //Innkommende Seriemelding
          case MT_INCOMING_ZAPP_PORT:

            // Sender innkommende seriemelding til HART node
                    Gateway_SendMessage_MASTER( msgPtr );

            break;

          case AF_INCOMING_MSG_CMD:
          case AF_INCOMING_GRP_MSG_CMD:
            // convert to incoming packet format
            MSGpkt = (afIncomingMSGPacket_t *)&(msgPtr[1]);

            // Process the incoming message
            MessageMSGCB( MSGpkt );

            // Release the data buffer
            osal_mem_free( MSGpkt->cmd.Data );
            break;

          case ZDO_NEW_DSTADDR:
            dstEP = msgPtr[ZDO_NEW_DSTADDR_DSTEP];
            dstAddr = (zAddrType_t *)&msgPtr[ZDO_NEW_DSTADDR_DSTADDR];
            ep = msgPtr[ZDO_NEW_DSTADDR_EP];
            // Change the endpoint into an index
            x = IndexFromEndpoint( ep );
            Gateway_DstAddr[x].addrMode = dstAddr->addrMode;
            Gateway_DstAddr[x].endPoint = dstEP;

            if ( dstAddr->addrMode == Addr16Bit )
              Gateway_DstAddr[x].addr.shortAddr = dstAddr->addr.shortAddr;
            else
            {
              osal_memcpy( Gateway_DstAddr[x].addr.extAddr,
                           dstAddr->addr.extAddr, Z_EXTADDR_LEN );
            }
            break;

          case ZDO_STATE_CHANGE:
            Gateway_NwkState = (devStates_t)msgPtr[1];
            if ( (Gateway_NwkState == DEV_ZB_COORD)
                 || (Gateway_NwkState == DEV_ROUTER)
                 || (Gateway_NwkState == DEV_END_DEVICE) )
            {}
            break;

          default:
            break;
        }

        // Release the memory
        osal_msg_deallocate( msgPtr );

        // Next
        msg = osal_msg_receive( Gateway_TaskID );
      }
    }

    // Send a message out - This event is generated by a timer
    //  (setup in Gateway_Init()).
    if ( events & GATEWAY_SEND_MSG_EVT )
    {
        for( x = 0; x < NUM_ENDPOINTS; x++ )
        {
          ep = Gateway_epDesc[x].endPoint;
        }

      }
}

/*********************************************************************
 * Event Generation Functions
```

```
 */
/*********************************************************************
 * @fn      HandleKeys
 *
 * @brief   Handles all key events for this device.
 *
 * @param   shift - true if in shift/alt.
 * @param   keys - bit field for key events. Valid entries:
 *                 EVAL_SW4
 *                 EVAL_SW3
 *                 EVAL_SW2
 *                 EVAL_SW1
 *
 * @return  none
 */
void HandleKeys( byte shift, byte keys )
{
    byte x;


  // Shift is used to make each button/switch dual purpose.
  if ( shift )
  {
    if ( keys & EVAL_SW1 )
    {
    }
    if ( keys & EVAL_SW2 )
    {
    }
    if ( keys & EVAL_SW3 )
    {
    }
    if ( keys & EVAL_SW4 )
    {
    }
  }
  else
  {
    if ( keys & EVAL_SW1 )
    {
          /* Send ut en testmelding på skjerm ved joystick opp */
          MT_SerialSendZAppMsg("TEST",4);

          /* Blink grønn diode */
          BlinkLed( LED1, 1, 90, 90 );
    }

    if ( keys & EVAL_SW2 )
    {
      // Initiate an End Device Bind Request for the mandatory endpoint
      for( x = 0; x < NUM_ENDPOINTS; x++ )
        ZDApp_SendEndDeviceBindReq( Gateway_epDesc[x].endPoint );
    }

    if ( keys & EVAL_SW3 )
    {

    }

    if ( keys & EVAL_SW4 )
    {
      // Initiate a Match Description Request (Service Discovery)
      //  for the mandatory endpoint
      for( x = 0; x < NUM_ENDPOINTS; x++ )
        ZDApp_AutoFindDestination( Gateway_epDesc[x].endPoint );
    }
  }
}

/*********************************************************************
 * LOCAL FUNCTIONS
 */

/*********************************************************************
 * @fn      MessageKVPCB
 *
 * @brief   Data message processor callback.  This function processes
 *          any incoming data - probably from other devices.  So, based
 *          on cluster ID, perform the intended action.
 *
 * @param   none
 *
 * @return  none
 */
void MessageKVPCB( afIncomingKVPPacket_t *pkt )
{

  switch ( pkt->clusterId )
  {
  }
```

```
}

/*********************************************************************
 * @fn      MessageMSGCB
 *
 * @brief   Data message processor callback.  This function processes
 *          any incoming data - probably from other devices.  So, based
 *          on cluster ID, perform the intended action.
 *
 * @param   none
 *
 * @return  none
 */
void MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
  switch ( pkt->clusterId )
  {

        /* Motatt HART melding fra Slave */
    case GATEWAY_SLAVE_CLUSTER_ID:

          /* Send motatt data ut på serieporten */
          MT_SerialSendZAppMsg(pkt->cmd.Data, pkt->cmd.DataLength);

      break;
  }
}

/*********************************************************************
 * @fn      Gateway_SendMessage_MASTER
 *
 * @brief   Sends the received serial message out over the air.
 *
 * @param   none
 *
 * @return  none
 */

void Gateway_SendMessage_MASTER( byte *msg )
{
  afStatus_t stat;

  // This message is setup to use APS ACKs (application layer acknowledgements)
  // APS ACKs take up bandwidth.  If you would like to increase the transmit
  // rate and you don't care about ACKing at the application, turn off APS ACK
  // by replacing AF_MSG_ACK_REQUEST with 0 in the function call below.
  stat = afFillAndSendMessage( &Gateway_DstAddr[0], Gateway_epDesc[0].endPoint,
                               GATEWAY_MASTER_CLUSTER_ID, 1, FRAMETYPE_MSG,
                               &Gateway_TransID, NULL, NULL, NULL, NULL,
                               msg[1], &msg[2],
                               AF_MSG_ACK_REQUEST, true, AF_DEFAULT_RADIUS );

  if ( stat != ZSuccess );
    // Wait to retry sending OTA message
    //osal_start_timer( SERIALAPP_MSG_HOLD_EVT, SERIALAPP_MSG_HOLD_TIMEOUT );
  else
    // Visual indication that message went out
    BlinkLed( LED1, 1, 90, 90 );
}

/*********************************************************************
 * @fn      IndexFromEndpoint
 *
 * @brief   Utility function that returns the index for the passed in endpoint.
 *
 * @param   endpoint - which endpoint
 *
 * @return  index into the endpoint descriptor
 *          NUM_ENDPOINTS if not found
 */
byte IndexFromEndpoint( byte endpoint )
{
  byte x;

  for ( x = 0; x < NUM_ENDPOINTS; x++ )
  {
    if ( endpoint == Gateway_epDesc[x].endPoint )
      break;
  }
  return x;
}
```

Gateway.h

```
#ifndef GATEWAY_H
#define GATEWAY_H
/*********************************************************************
    Copyright (c) 2005 by Figure 8 Wireless, Inc., All Rights Reserved.
    Permission to use, reproduce, copy, prepare derivative works,
    modify, distribute, perform, display or sell this software and/or
    its documentation for any purpose is prohibited without the express
    written consent of Figure 8 Wireless, Inc.
*********************************************************************/

#ifdef __cplusplus
extern "C"
{
#endif

/*********************************************************************
 * INCLUDES
 */
#include "ZComDef.h"

/*********************************************************************
 * CONSTANTS
 */

#define GATEWAY_SEND_MSG_TIMEOUT   5000
#define GATEWAY_SEND_MSG_EVT       0x0001
#define GATEWAY_FLAGS              0

#define GATEWAY_ENDPOINT_1 1
#define GATEWAY_PROFID 2
#define GATEWAY_DEVICEID 1
#define GATEWAY_DEVICE_VERSION 1
#define GATEWAY_MAX_OUT_CLUSTERS 1
#define GATEWAY_MAX_IN_CLUSTERS 1
#define GATEWAY_MASTER_CLUSTER_ID 1
#define GATEWAY_SLAVE_CLUSTER_ID 2

/*
 * Task Initialization for the Generic Application
 */
extern void Gateway_Init( byte task_id );

/*
 * Task Event Processor for the Generic Application
 */
extern void Gateway_ProcessEvent( byte task_id, UINT16 events );

/*********************************************************************
*********************************************************************/

#ifdef __cplusplus
}
#endif

#endif /* GATEWAY_H */
```

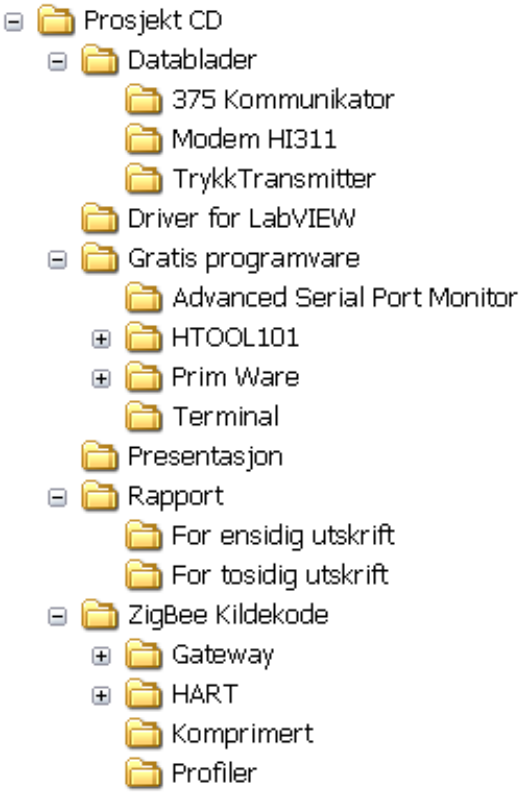
OSAL_Gateway.c

```
/**************************************************************************
  Copyright (c) 2005 by Figure 8 Wireless, Inc., All Rights Reserved.
  Permission to use, reproduce, copy, prepare derivative works,
  modify, distribute, perform, display or sell this software and/or
  its documentation for any purpose is prohibited without the express
  written consent of Figure 8 Wireless, Inc.
 **************************************************************************
 **************************************************************************
 * INCLUDES
 */
#include "ZComDef.h"
#include "OSAL.h"
#include "OSAL_Tasks.h"
#include "OSAL_Custom.h"

#if defined ( MT_TASK )
  #include "MTEL.h"
#endif

#if !defined( NONWK )
  #include "nwk.h"
  #include "APS.h"
  #include "ZDApp.h"
#endif

#include "Gateway.h"
#include "OnBoard.h"


/**************************************************************************
 * FUNCTIONS
 **************************************************************************/

/**************************************************************************
 * @fn      osalAddTasks
 *
 * @brief   This function adds all the tasks to the task list.
 *          This is where to add new tasks.
 *
 * @param   void
 *
 * @return  none
 */
void osalAddTasks( void )
{
#if defined( MT_TASK )
  osalTaskAdd( MT_TaskInit, MT_ProcessEvent );
#endif

  osalTaskAdd( OnBoard_TaskInit, OnBoard_ProcessEvent );

  osalTaskAdd( nwk_init, nwk_event_loop );
  osalTaskAdd( APS_Init, APS_event_loop );
  osalTaskAdd( ZDApp_Init, ZDApp_event_loop );
  osalTaskAdd( Gateway_Init, Gateway_ProcessEvent );
}
```

Tillegg D

CD

Figur D.1 viser mappefordelingen på vedlagte CD.



Figur D.1: Filtre for vedlagt cd.

Bibliografi

[1] ZigBee Alliance. '*ZigBee Specification*'. ZigBee Document 053474r06, Version 1.0, 14 Desember , 2004.

[2] Chipcon. '*ProfileBuilder Getting Started Guide*'. Document Number: F8W-2004-0010, april 2005.

[3] William C. Craig '*Zigbee: Wireless Control That Simply Works*'. http://www.zigbee.org/.

[4] Venkat Bahl. '*ZigBee Overview*'. http://www.zigbee.org/en/resources/zigbeeoverview4.pdf, september 2002.

[5] IEEE Computer Society. 'Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)', IEEE standard, 2003.

[6] HART Communication Foundation. http://www.hartcomm.org.

[7] HCF. 'HART BASIC course'. http://hartcomm.org/etraining/courses.php, 2005.

[8] Analog Services. '*Online HART book*'. http://www.analogservices.com/about_part0.htm, septemper 1999.

[9] Romilly's HART® and Fieldbus Web Site. http://www.romilly.co.uk/

[10] E. Undheim.'En evaluering av standardene IEEE 802.15.4 og ZigBee for indistruelle anvendelser'. Masteroppgave ved NTNU, våren 2005.

[11] Perry Sink. '*A Comprehensive Guide to Industrial Networks*'. http://www.sensorsmag.com/articles/0601/28/index.htm, *Sensors Magazine Online*, juni 2001.

[12] Ed T. Ladd, Jr. '*Process Engineering: HART - Dispelling the Myths*'. http://www.chemicalprocessing.com/articles/2005/64.html, *Chemical Processing Magazine*, 2005.

[13] Dick Johnson. '*The basics of HART*'. http://www.manufacturing.net/ctl/article/CA186130, *Control Engineering Europe*, November 2000.

[14] Dag Vegard Tveitå. 'Trådløse sensornettverk'. Masteroppgave ved NTNU, våren 2005.

[15] Cato Andre Jensen. 'Trådløse sensornettverk'. Masteroppgave ved NTNU, våren 2005.

[16] Søkemaskin: http://www.google.com.

[17] Microflex. '*InLink HART Modem Module*'. http://www.microflx.com/InLinkModem.htm

[18] Elfa. http://www.elfa.se/no/

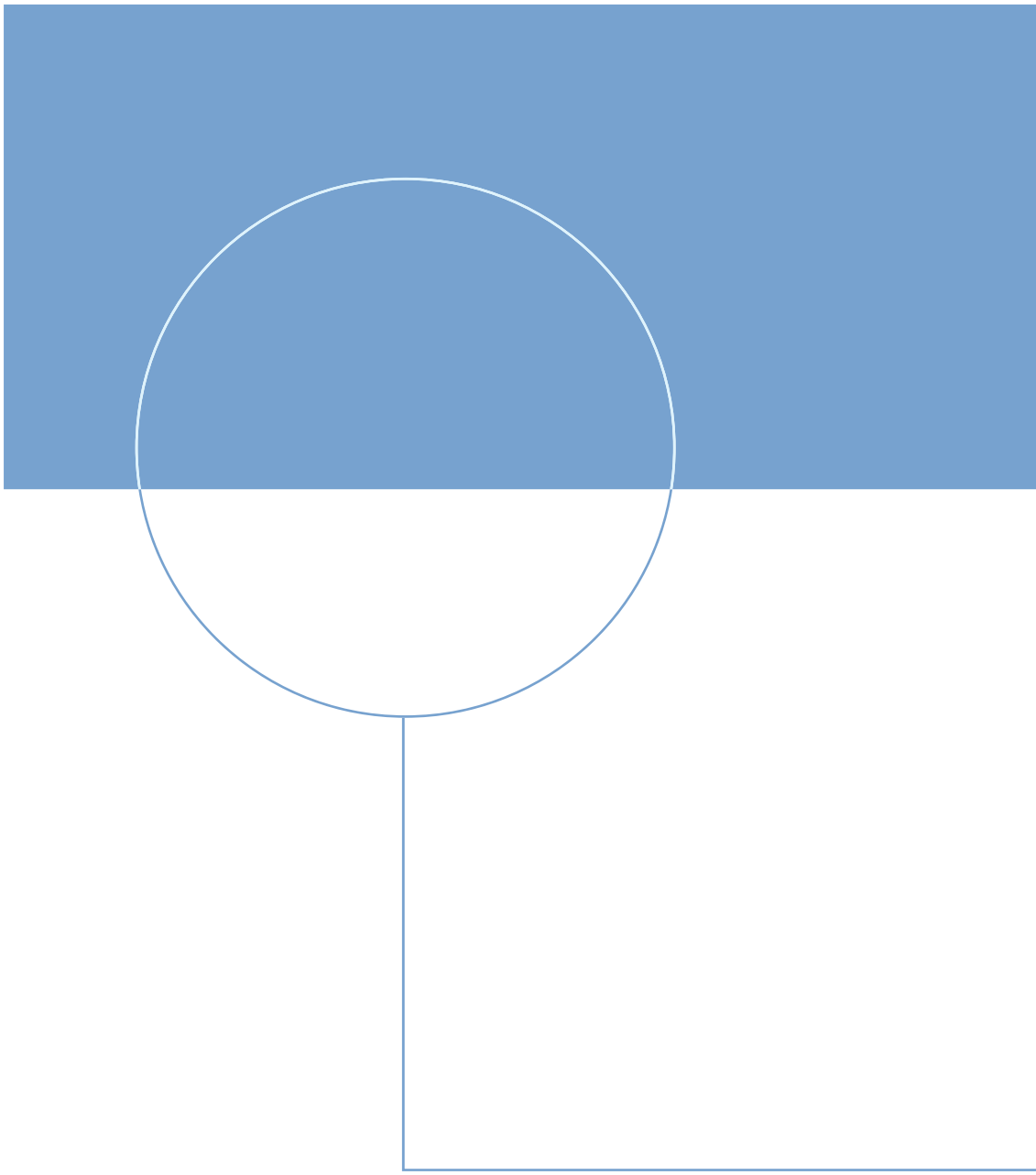
[19] Tor Onshus. 'Instrumenteringssystemer'. NTNU, januar 1997.

[20] Triangle Micro Solutions. 'Free HART Software'. http://www.trianglemicro.com/hart.htm, 2003.

[21] Romilly Bowden. 'HART software'. http://www.romilly.co.uk/software.htm, 1998 - 2005.

[22] Bray++. 'Terminal'. http://bray.velenje.cx/avr/terminal/, 1997 - 2004.

[23] AGG Software '*Advanced Serial Port Monitor*'. http://www.aggsoft.com/serial-port-monitor.htm, 1999 - 2005.



NTNU
Det skapende universitet