NTNU
Det skapende universitet

Programvaredefinert radio
Mulige hyllevareløsninger for DSRC-anvendelser

Einar Thorsrud

Master i elektronikk
Oppgaven levert: Juni 2009
Hovedveileder: Lars Magne Lundheim, IET

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Oppgavetekst

Radiokommunikasjonsfeltet er i en rivende utvikling mot generelle elektronikkplattformer hvor funksjonaliteten er gitt av programvare. Programvaredefinert radio er et konsept som sikter mot en idealradio som kan programmeres til å fungere som en hvilken som helst annen radio. Fremveksten av standardiserte maskinvare- og programvareplattformer for programvareradioer er et skritt på veien mot dette målet.

Oppgaven innebærer å analysere to frie rammeverk for programvaredefinert radio, GNU Radio og «Open Source SCA Implementation::Embedded». Undersøkelsen skal drøfte hvordan de to rammeverkene kan benyttes sammen med radioplattformen «Universal Software Radio Peripheral» (USRP), i forbindelse med dedikert kortholdslink (DSRC), som brukes til elektronisk bompengeinnkreving.

Målet med oppgaven er å implementere deler av fysisk lag i en veikantenhet for DSRC-kommunikasjon som en programvaredefinert radio ved å benytte hyllevarer. Det bør undersøkes om det er mulig å integrere realiseringen i en kompakt innvevd enhet basert på USRP og datamaskinkomponenter.

Oppgaven gitt: 15. januar 2009
Hovedveileder: Lars Magne Lundheim, IET

Programvaredefinert radio
Mulige hyllevareløsninger for DSRC-anvendelser

Einar Thorsrud

Juni 2009

Sammendrag

Utviklingen innen radiokommunikasjon går i retning av rekonfigurerbare *programvaredefinerte radioer*, der funksjonaliteten og signalbehandlingen er implementert i programvare. Fremveksten av standardiserte maskinvareplattformer og programvarerammeverk, som gjør det mulig å utvikle nye og fleksible radioer ved hjelp av hyllevarer, er et resultat av dette.

To sentrale frie rammeverk for programvaredefinert radio er: GNU Radio, et selvstendig rammeverk; og «Open Source SCA Implementation::Embedded», en fri implementering av «Software Communications Architecture».

Denne masteroppgaven undersøker hvordan radioplattformen «Universal Software Radio Peripheral» (USRP) kan brukes sammen med de to frie rammeverkene til å realisere en veikantenhet for elektronisk bompengeinnkreving. Det innledende litteraturstudiet viser at GNU Radio er best egnet til realiseringen på grunn av lav interkomponentlatens, et stort utvalg ferdige signalbehandlingsblokker og god støtte for USRP.

Gjennom arbeidet med denne oppgaven ble det implementert en sender og en koherent mottaker til en veikantenhet ved hjelp av GNU Radio og USRP. Realiseringene fungerer delvis, men det oppsto betydelige problemer under testing. Beregningskompleksiteten er lav nok til at det vil være mulig å pakke radiofunksjonaliteten inn i en integrert enhet basert på datamaskinkomponenter og USRP.

Forord

Denne masteroppgaven er utført ved Norges teknisk-naturvitenskapelige universitet som en del av graden Master i teknologi/sivilingeniør i elektronikk. Oppgaven er skrevet under veiledning av førsteamanuensis Lars Lundheim fra gruppen for signalbehandling ved Institutt for elektronikk og telekommunikasjon, og Torstein Dybdahl fra Q-Free ASA i Trondheim.

Jeg vil takke mine veiledere Lars Lundheim og Torstein Dybdahl for verdifulle tilbakemeldinger gjennom hele prosjektet. Jeg er også takknemlig overfor Helton Kosturi, som har bidratt med praktisk hjelp i forbindelse med forsøk hos Q-Free. Deltakerene på e-postlistene til GNU Radio og OSSIE fortjener en takk for sine innsiktsfulle råd og svar på mine spørsmål. Til slutt skylder jeg Øyvind og min samboer Ingeborg en stor takk for møysommelig korrekturlesing.

Trondheim, juni 2009

Einar Thorsrud

Innhold

1 Introduksjon **1**
- 1.1 Definisjoner 2
- 1.2 Oppbygning av rapporten 2

2 Programvaredefinert radio **4**

3 Signal- og kommunikasjonsteori **6**
- 3.1 Binær faseskiftnøkling 6
- 3.2 Differensiell binær faseskiftnøkling 7
- 3.3 Amplitudeskiftnøkling 8
- 3.4 Linjekoding 11
- 3.5 Pulsforming 11
- 3.6 Bærebølgegjenvinning med Costas-sløyfe 12
- 3.7 Taktgjenvinning med mM&M-algoritmen 13
- 3.8 Digital frekvenskonvertering 14
- 3.9 Endring av punkprøvingsrate 14

4 Radioplattformen USRP **16**
- 4.1 Analog-digital- og digital-analog-omforming 18
- 4.2 Endring av punktprøvingsrate i FPGA 18
- 4.3 Båndbreddebegrensninger 19
- 4.4 Forsinkelse i USB 19
- 4.5 Datterkort 20

5 Frie rammeverk for SDR **22**
- 5.1 GNU Radio 22
  - 5.1.1 Arkitektur 22
  - 5.1.2 Blokker 24
  - 5.1.3 Flytskjemamekanismer 25
  - 5.1.4 Flytskjemaer 25

5.1.5 Pakkebehandling 26
5.2 SCA og OSSIE 26
5.2.1 Strukturen i SCA 27
5.2.2 OSSIE 29
5.3 Sammenligning av GNU Radio og OSSIE 29
5.3.1 Kompleksitet og ytelse 30
5.3.2 Interkomponentlatens 31
5.3.3 Oppsummering 32

6 Dedikert kortholdslink 33
6.1 Fysisk lag ved 5,8 GHz 34
6.1.1 Nedlink 35
6.1.2 Opplink 35
6.2 Mediumtilgang og logisk linkkontroll 36
6.2.1 Rammestruktur 37
6.2.2 Mediumtilgangskontroll 38
6.3 Applikasjonslaget 38

7 Realisering av en veikantenhet 39
7.1 Valg av rammeverk 39
7.2 Utstyr 40
7.3 Sender 40
7.3.1 Sender implementert med GNU Radio 41
7.3.2 Sender implementert med OSSIE 42
7.4 Mottaker 43
7.4.1 Koherent mottaker med GNU Radio 43
7.4.2 Alternativ ikke-koherent mottaker 46
7.5 Test og verfisering 47
7.5.1 Senderen implementert med GNU Radio 47
7.5.2 Senderen implementert med OSSIE 49
7.5.3 Koherent mottaker 49
7.6 Beregningskompleksitet 50

8 Konklusjon 51

Bibliografi 53

A Kildekode 58
A.1 Senderen realisert med GNU Radio 58
A.2 Mottakeren realisert med GNU Radio 61
A.3 Pulsformer-blokken i GNU Radio 65

Register 70

Figurer

Forsidebilde: «Babels tårn» av Pieter Brueghel den eldre, ca. 1563

2.1 Programvaredefinert radio. 4

3.1 Blokkdiagram for en BPSK-sender og en koherent mottaker. . 7
3.2 Blokkdiagram for en DBPSK-sender og en mottaker. 8
3.3 Signalromdiagram for et mottatt DBPSK-signal. 9
3.4 Bitfeilrate for BPSK og DBPSK. 9
3.5 Blokkdiagram for en generell ASK-sender. 10
3.6 Bifaselinjekoden FM0. 11
3.7 Linjekoden NRZI. 11
3.8 Costas-sløyfe for bærebølgegjennvining. 12
3.9 Modifisert Mueller og Müller algoritme for QPSK 13
3.10 CIC-filter. 15

4.1 Grunnleggende oppbygning av USRP med datterkort. 17
4.2 Hovedkortet til USRP med standard FPGA-programvare. . . 17
4.3 Overføring av punktprøver mellom en datamaskin og USRP. . 19
4.4 USRP med LFTX- og LFRX-datterkortene. 21

5.1 Et SDR-system med GNU Radio og USRP. 23
5.2 Prinsipiell lagdeling i JTRS SCA. 27
5.3 Detaljert fremstilling av SCA OE. 28

6.1 Protokollstakken i DSRC. 34
6.2 Rammestruktur for fysisk lag i DSRC. 35
6.3 Frekvensspektrum for den laveste DSRC-kanalen. 35
6.4 Kommunikasjonssonen er i hovedsak foran antennen. 36
6.5 Rammestrukturen i DSRC. 37

7.1 Generell DSRC RSU-sender. 41
7.2 Senderen slik den er realisert med GNU Radio og USRP. . . . 41

7.3 Basisbåndsignalet når datasekvensen 01010 er FM0-kodet ASK. 42
7.4 Senderen realisert i OSSIE. 43
7.5 Generell koherent mottaker. 44
7.6 Deteksjon og synkronisering i denne realiseringen. 44
7.7 Basisbåndsignalet fra USRP målt med oscilloskop. 48
7.8 Utsendt passbåndsignal nedkonvertert til en mellomfrekvens. . 48

Tabeller

5.1 Sammenligning mellom GNU Radio og OSSIE. 32

7.1 Maskinvare som er benyttet i den praktiske realiseringen. . . . 40
7.2 Programvare som er benyttet. 40
7.3 Beregningskompleksitet i senderen realisert med GNU Radio. 49
7.4 Beregningskompleksitet i mottakeren realisert med GNU Radio. 50

Forkortelser

ADC	Analog-digital-omformer
ASK	Amplitudeskiftnøkling
BER	Bitfeilrate
BPSK	Binær faseskiftnøkling
BST	«Beacon Service Table»
CF	«Core Framework»
CIC	«Cascaded integrator-comb»
CORBA	«Common Object Request Broker Architecture»
CPU	Hovedprosessor
DAC	Digital-analog-omformer
DBPSK	Differensiell binær faseskiftnøkling
DC	Likespenning
DPSK	Differensiell faseskiftnøkling
DSP	Digital signalprosessor
DSRC	Dedikert kortholdslink
EN	Europeisk standard
FIFO	Først inn – først ut
FIR	Endelig impulsrespons
FPGA	Rekonfigurerbar logikk
GNU	«GNU is not Unix»

GPL	«GNU General Public License»
GPP	Generell prosessor
GRC	«GNU Radio Companion»
GSM	Globalt System for Mobilkommunikasjon
IIR	Uendelig impulsrespons
ISM	«Industrial, Scientific and Medical»
JTRS	«Joint Tactical Radio System»
LID	«Link Identifier»
LLC	Logisk linkkontroll
LPDU	«Link layer Protocol Data Unit»
LPF	Lavpassfilter
LSB	Minst signifikante bit
MAC	Mediumtilgangskontroll
MDR	Medium datarate
MIMO	«Multiple Input, Multiple Output»
mM&M	Modifisert Mueller & Müller
NS	Norsk Standard
OBU	Bilbrikke («On Board Unit»)
OEF	«OSSIE Eclipse Feature»
ORB	«Object Request Broker»
OSI	«Open Systems Interconnection»
OSSIE	«Open Source SCA Implementation::Embedded»
PC	Personlig datamaskin
POSIX	«Portable Operating System Interface»
PSK	Faseskiftnøkling
QPSK	Kvadratur faseskiftnøkling
RF	Radiofrekvens
RSU	Veikantenhet («Roadside Unit»)
RTTT	Veitransporttelematikk

SCA	«Software Communications Architecture»
SDR	Programvaredefinert radio
SMA	«SubMiniature version A» – En kontakt for koaksiale RF-kabler
SNR	Signal-støy-forhold
SWIG	«Simplified Wrapper and Interface Generator»
USB	«Universal Serial Bus»
USRP	«Universal Software Radio Peripheral»
VCO	Spenningsstyrt oscillator
XML	«Extensible Markup Language»

Kapittel 1

Introduksjon

Den raske utviklingen innen prosesseringshastighet har siden digitalteknikkens begynnelse åpnet for stadig nye bruksområder for datamaskiner og andre prosesseringsenheter. Programvaredefinert radio (SDR) er en idé som har vokst frem som følge av de mulighetene som ligger i å utføre kompleks sanntidssignalbehandling på generelle prosesseringsplattformer. Programvaredefinert radio sikter mot en fleksibel idealradio som kan erstatte en hvilken som helst annen radio. Fremveksten av standardiserte maskinvareplattformer og programvarerammeverk for SDR er et steg på veien mot dette målet.

Denne rapporten beskriver arbeidet som er utført i forbindelse med en masteroppgave i signalbehandling og kommunikasjon ved Norges teknisk-naturvitenskapelige universitet, på oppdrag fra Q-Free ASA. Undersøkelsen tar utgangspunkt i å vurdere to frie rammeverk for programvaredefinert radio, GNU Radio og «Open Source SCA Implementation::Embedded» (OSSIE). Rammeverkene vurderes opp mot hverandre for å undersøke hvor godt egnet de er til anvendelser innen dedikert kortholdslink (DSRC). Målet er å vurdere muligheter for å benytte standardiserte SDR-løsninger til å realisere en DSRC-veikantenhet for elektronisk bompengeinnkreving.

En veikantenhet blir delvis realisert ved hjelp av hyllevarekomponenter og radioplattformen «Universal Software Radio Peripheral» (USRP). På bakgrunn av realiseringen blir det drøftet hvordan USRP og hyllevare datamaskinkomponenter kan benyttes til en kompakt og fleksibel integrert veikantenhet.

GNU Radio blir behandlet i større detalj enn OSSIE i denne rapporten, ettersom OSSIE ble grundig behandlet i et fordypningsprosjekt utført av undertegnede høsten 2008 [1].

1.1 Definisjoner

Denne delen gir en kort introduksjon til de mest sentrale begrepene som er brukt i rapporten. De blir grundigere utdypet i de påfølgende kapitlene.

En *programvaredefinert radio* (SDR) er en *rekonfigurerbar* radio, der funksjonaliteten er definert i programvare.

I programvaredefinert radio er en *bølgeform* en beskrivelse av hele funksjonaliteten til radioen, fra modulasjon til høyere lags protokoller.

«Universal Software Radio Peripheral» (USRP) er en rimelig maskinvareplattform for programvaredefinert radio.

GNU Radio er et fritt tilgjengelig rammeverk for programvaredefinert radio.

«Software Communications Architecture» (SCA) er en åpen arkitektur for programvaredefinert radio.

«Open Source SCA Implementation::Embedded» (OSSIE) er en fri implementering av SCA.

Dedikert kortholdslink (DSRC) er en standard for kortholdskommunikasjon som brukes til veitransporttelematikk, blant annet elektronisk bompengeinnkreving.

1.2 Oppbygning av rapporten

Denne masteroppgaven er delt inn i 8 kapitler:

Kapittel 1 er en introduksjon til oppgaven og rapporten.

Kapittel 2 gir en generell oversikt over programvaredefinert radio.

Kapittel 3 introduserer nødvendig bakgrunnsteori i signalbehandling og kommunikasjon.

Kapittel 4 presenterer radioplattformen USRP.

Kapittel 5 gir en oversikt over GNU Radio og OSSIE, og vurderer dem opp mot hverandre.

Kapittel 6 gir en grunnleggende oversikt over DSRC-standardene.

Kapittel 7 beskriver hvordan fysisk lag i en veikantenhet for DSRC ble delvis realisert i GNU Radio og OSSIE sammen med en USRP.

Kapittel 8 presenterer resultatene og konklusjonene som kan trekkes fra arbeidet med en programvaredefinert realisering av en veikantenhet

for dedikert kortholdslink, og kommer med vurderinger og forslag til videre arbeid.

Norske faguttrykk er forsøkt brukt så godt det lar seg gjøre, men der det har vært vanskelig å finne allment aksepterte norske termer, brukes i hovedsak de engelske.

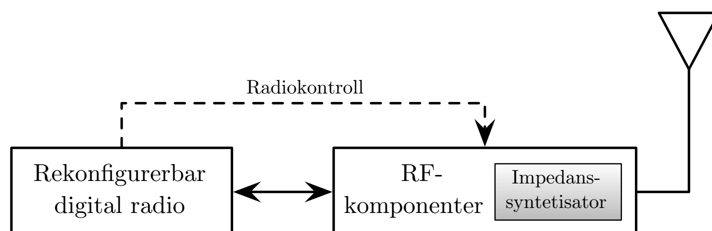
Kapittel 2

Programvaredefinert radio

Ideell programvaredefinert radio (SDR) er en *Babels tårn-teknologi* [2]. Dette innebærer at trådløse enheter som tidligere bare forsto ett eller flere språk eller standarder vil kunne kommunisere med hverandre på en hvilken som helst frekvens eller protokoll. Moderne digitale radioer er generelt programvaredefinerte i den forstand at mye av funksjonaliteten er implementert i programvare på en digital prosesseringsplattform. Den vesentlige forskjellen mellom tradisjonelle programvarebaserte radioer og SDR er muligheten til å endre funksjonaliteten *etter* produksjonstidspunktet [3].

Uttrykket *programvaredefinert radio* tilskrives Joseph Mitola, og hans artikkel [4] fra 1992. SDR blir sett på som en viktig forutsetning for kognitiv radio [5]. Tidlig på 90-tallet begynte det amerikanske forsvarsdepartementet prosjektet «SpeakEasy», som hadde som mål å støtte ti militære bølgeformer i en enkel enhet [6]. Det amerikanske forsvaret hadde allerede da over to tiår med forskning på programmerbare radioer bak seg. Den første programmerbare radioen, «Integrated Communications Navigation Identification Avionics» (ICNIA), ble demonstrert i 1987 [7].

En ideell SDR-plattform er et system der all prosessering utføres i program-



Figur 2.1: Programvaredefinert radio.

vare, og hvor alle nødvendige analoge radiokomponenter kan justeres i programvare [8], som vist i figur 2.1. Dette innebærer også at en ideell programvaredefinert radio skal kunne bruke helt forskjellige frekvensbånd uten endringer i maskinvaren eller radiokomponentene. Et slikt krav er vanskelig å tilfredsstille uten store kostnader og høy kompleksitet, og det er på dette punktet de fleste praktiske SDR-systemene skiller seg fra det ideelle. Desto nærmere AD/DA-omformeren flyttes antennen, desto nærmere er radioen en ideell programvaredefinert radio [6].

En praktisk definisjon av en realiserbar SDR kan være:

> En programvaredefinerte radio er et element i et trådløst nettverk der funksjonsmåte og parametre kan endres *etter* produksjonstidspunktet ved hjelp av programvare [9].

En programvaredefinert radio vil ofte, men ikke nødvendigvis, bestå av en eller flere forskjellige prosesseringsenheter, blant annet generelle prosessorer (GPP), digitale signalprosessorer (DSP) og rekonfigurerbar logikk (FPGA) [8].

Med stadig økende datakraft, jamfør Moores lov [10] og nye billige plattformer, vil programvaredefinert radio basert på PC-arkitektur kunne være et godt alternativ for små serier, samt forskning og utvikling. I 2008 ble det eksempelvis laget en portabel programvaredefinert radio, basert på USRP og et lite hovedkort med en GPP og en trykkfølsom skjerm [11].

For å oppnå større kompatibilitet og lavere priser innen SDR-markedet, har det amerikanske forsvaret gjennom prosjektet «Joint Tactical Radio System» (JTRS) utviklet «Software Communications Architecture» (SCA), som er blitt en ledende programvarearkitektur innen programvaredefinert radio.

Kapittel 3

Signal- og kommunikasjonsteori

Dette kapitlet beskriver den nødvendige signalbehandlingen og kommunikasjonsteorien for å implementere fysisk lag i en DSRC-sender -og mottaker. Metodene som er beskrevet brukes enten i maskinvareplattformen USRP, som beskrives senere i kapittel 4, eller i DSRC-standardene eller radiorammeverkene som er undersøkt og brukt i denne oppgaven.

3.1 Binær faseskiftnøkling

Binær faseskiftnøkling (BPSK) er en modulasjonsmetode hvor informasjonen ligger i fasen til børebølgen. De binære symbolene 1 og 0 representeres av henholdsvis $s_1(t)$ og $s_2(t)$ [12]. Disse kan defineres som

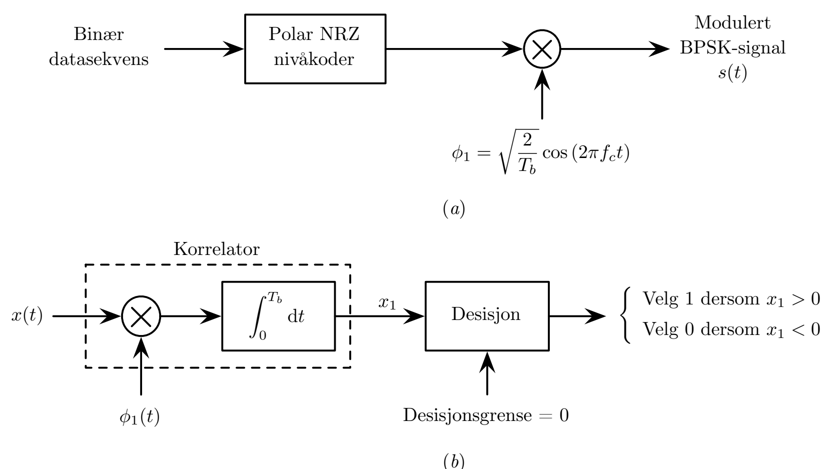
$$s_1(t) = \sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t) \tag{3.1}$$

$$s_2(t) = \sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t + \pi) = -\sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t), \tag{3.2}$$

hvor E_b er signalenergien per bit, f_c er bærebølgefrekvensen, T_b er varigheten av et bit og $0 \leq t \leq T_b$.

Figur 3.1 viser blokkskjemaer for en BPSK sender (a) og en koherent BPSK-mottaker (b). Senderen koder de binære dataene til en konstant amplitude på henholdsvis $+\sqrt{E_b}$ eller $-\sqrt{E_b}$ for 1 eller 0 i en polar NRZ-koder. Dette signalet mikses med en sinusformet bærebølge $\phi_1(f)$, og produktet $s(t)$ er det modulerte BPSK-signalet.

For å detektere den originale binære sekvensen av enere og nuller blir det mottatte BPSK-signalet $x(t)$ med kanalstøy korrelert med et lokalt generert



Figur 3.1: Blokkdiagram for en BPSK-sender (a) og en koherent mottaker (b) [12].

referansesignal $\phi_1(t)$, som vist i figur 3.1 (b). Dersom signalet x_1 er større enn terskelverdien, tas det en avgjørelse om at det mottatte symbolet er 1, og i motsatt fall 0 dersom signalet er under terskelverdien.

Bitfeilraten til BPSK i en kanal med additiv hvit gaussisk støy er

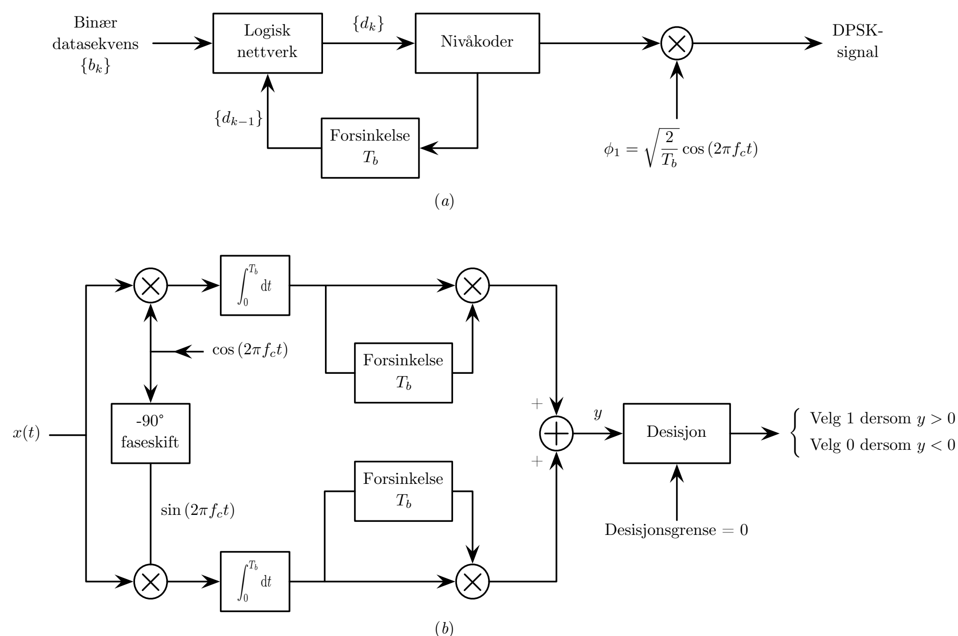
$$P_b = \mathrm{Q}\left(\sqrt{\frac{2E_b}{N_0}}\right) = \frac{1}{2}\,\mathrm{erfc}\left(\sqrt{\frac{E_b}{N_0}}\right), \tag{3.3}$$

hvor erfc() er den komplementære feilfunksjonen.

3.2 Differensiell binær faseskiftnøkling

Metoden for BPSK-modulasjon beskrevet over krever at mottakeren er fasesynkronisert mot det innkommende signalet. For å slippe dette kan senderen benytte differensiell koding, noe som gjør mottak og deteksjon enklere ettersom mottakeren kun trenger å forholde seg til endring fra et symbol til et annet [12]. Differensiell binær faseskiftnøkling (DBPSK) kan ses på som ren BPSK, men der dataene er differensielt kodet slik at en endring fra foregående bit representeres med 0 og ingen endring med 1, eller visa versa. Figur 3.2 viser prinsippet til en DBPSK-sender (a) og en DPSK-mottaker (b).

DBPSK-senderen har en tilbakekobling av forrige symbol d_{k-1} til et logisk nettverk som opererer slik at dersom det innkommende binære symbolet b_k er 1, forblir symbolet d_k uendret i forhold til det forrige bitet. Dersom det innkommende binære symbolet b_k er 0, endres symbolet d_k med tanke på det forrige bitet. DBPSK innebærer altså kun en differensiell *prekoding* av



Figur 3.2: Blokkdiagram for en DBPSK-sender (a) og en ikke-koherent mottaker (b) [12].

dataene før de moduleres, og resten av senderen er identisk med BPSK-senderen.

Under mottak av DBPSK trenger ikke fasen til det innkommende signalet å være kjent. Ved å bruke et signalrom med en fasekomponent (I) og en 90° forskjøvet kvadraturfasekomponent (Q), vil de mulige signalpunktene være $(A\cos\theta, A\sin\theta)$ og $(-A\cos\theta, -A\sin\theta)$. Her er A amplituden og θ den ukjente fasen til signalet $x(t)$. Mottakeren måler koordinatene (x_{I_0}, x_{Q_0}) ved tiden $t = T_b$ og (x_{I_1}, x_{Q_1}) ved tiden $t = 2T_b$, vist i figur 3.3. Det må da avgjøres om de to punktene tilsvarer det samme punktet eller ikke.

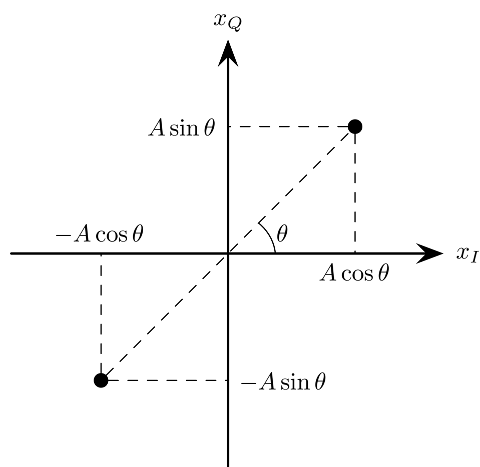
Bitfeilraten til DBPSK i en kanal med additiv hvit gaussisk støy er gitt av

$$P_b \approx \frac{1}{2}e^{-\frac{E_b}{N_0}}. \tag{3.4}$$

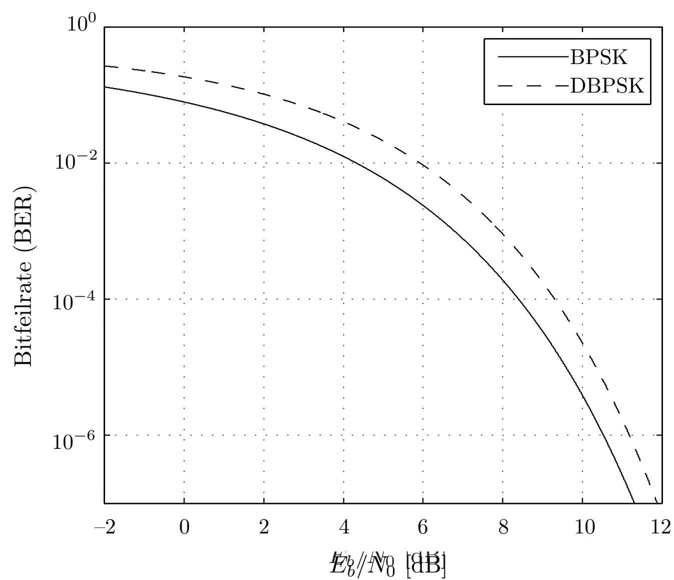
Sammenhengen mellom bitfeilrate og bitenergi over støyenergi E_b/N_0 for BPSK og DBPSK er vist i figur 3.4.

3.3 Amplitudeskiftnøkling

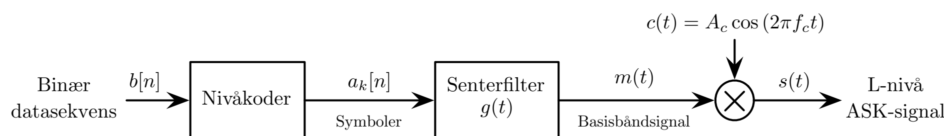
Digital amplitudeskiftnøkling (ASK) er en form for amplitudemodulasjon der informasjonen ligger i forskjellige diskrete nivåer på amplituden, og det er



Figur 3.3: Signalromdiagram for et mottatt DBPSK-signal.



Figur 3.4: Bitfeilrate for BPSK og DBPSK.



Figur 3.5: Blokkdiagram for en generell ASK-sender.

$\log_2 L$ bit per symbol, hvor L er antall diskrete nivåer. Denne gjennomgangen tar for seg tradisjonell amplitudemodulasjon med dobbelt sidebånd, ettersom det er brukt i DSRC. Prinsippet for senderen er vist i figur 3.5.

Informasjonssignalet moduleres på en bærebølge

$$c(t) = A_c \cos(2\pi f_c t), \tag{3.5}$$

hvor f_c er frekvensen og A_c er amplituden til bærebølgen. Dersom $m(t)$ er et informasjonsbærende basisbåndsignal og k_a er amplitudefølsomheten i modulatoren, er det amplitudemodulerte signalet [12]

$$s(t) = A_c [1 + k_a m(t)] \cos(2\pi f_c t). \tag{3.6}$$

Når A_m er amplituden til basisbåndsignalet er modulasjonsindeksen er gitt av forholdet

$$\beta = \frac{k_a A_m}{A_c}. \tag{3.7}$$

Dersom $|k_a m(t)| < 1$ og bærebølgefrekvensen f_c er mye større enn den høyeste frekvenskomponenten W i $m(t)$, har envelopen til $s(t)$ en fasong som tilsvarer basisbåndsignalet $m(t)$. Der modulasjonindeksen $\beta < 1$ vil bærebølgen ikke bli undertrykt, noe som gir en spektrallinje ved f_c. Dette innebærer bortkastet effekt som ikke gir bedre støymargin, men gjør det mulig å lage svært enkle mottakere [12].

Dersom $g(t)$ er en nyquistpuls og det ikke er noen intersymbolinterferens er sannsynligheten for feil er gitt av

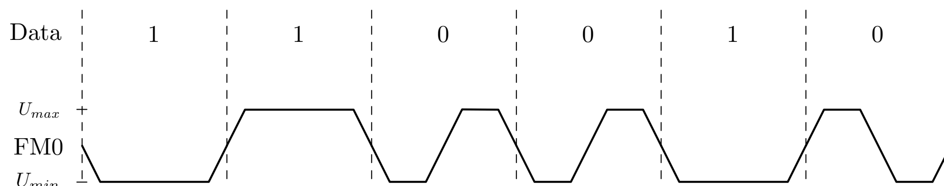
$$P_e = \left(1 - \frac{1}{L}\right) \operatorname{erfc}\left(\frac{Ag(0)}{\sqrt{2}\,(L-1)\,\sigma_N}\right), \tag{3.8}$$

hvor L er antallet nivåer som brukes i overføringen, A er den maksimale signalspenningen, og σ_N er additiv hvit gaussisk støy. Dersom $L = 2$ og $g(0) = 1$ er bitfeilsannsynligheten

$$P_b = \frac{1}{2} \operatorname{erfc}\left(\frac{A}{\sqrt{2}\,\sigma_N}\right). \tag{3.9}$$

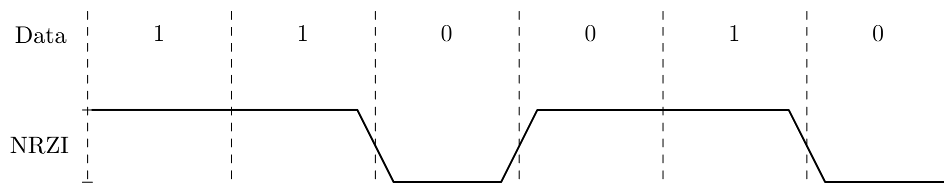
3.4 Linjekoding

FM0 er en bifaselinjekode som har en transisjon ved begynnelsen av hvert bit. En binær 1 representeres av en transisjon ved begynnelsen og slutten av symbolet, og en binær 0 representeres ved en ekstra transisjon i midten av symbolet. Se figur 3.6.



Figur 3.6: Bifaselinjekoden FM0. Her vist med U_{max} og U_{min}, som har sammenheng med modulasjonsindeksen beskrevet i forbindelse med realiseringen av senderen i kapittel 7.3.1.

I differensielle modulasjonsformer foregår prekodingen som en form for linjekode, og i differensiell PSK brukes NRZI («Non-Return-to-Zero Inverted»). Denne er vist i figur 3.7. Her er binær 0 representert som en transisjon i det fysiske nivået, mens binær 1 representeres med ingen endring fra det foregående symbolet.



Figur 3.7: Linjekoden NRZI.

3.5 Pulsforming

Pulsforming brukes for å oppnå ønskede spektrale egenskaper, der pulsene som skal sendes, filtreres med et senderfilter i basisbånd før miksing. Det ideelle filteret som gir best spektrale egenskaper, produserer en nyquistpuls. Dette er imidlertid ikke realiserbart. Det er også dårlig med tanke på støymargin dersom det ikke punktprøves på optimale tidspunkt i mottakeren.

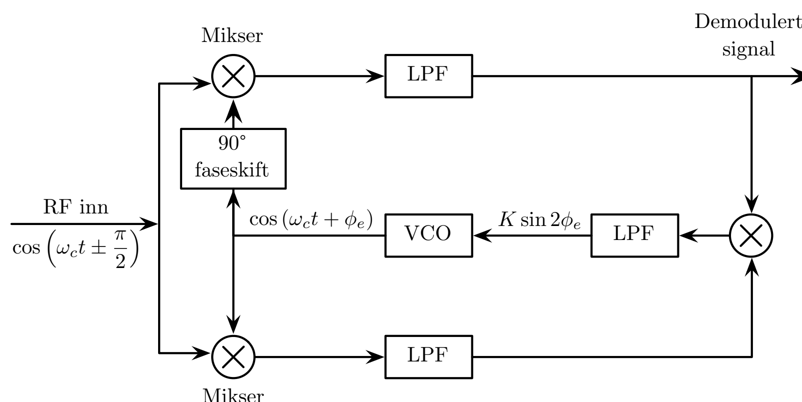
3.6 Bærebølgegjenvinning med Costas-sløyfe

Synkronisering er en sentral del av radiomottakere, og frekvens- og fasesynkronisering er en forutsetning for koherent deteksjon. Bærebølgegjenvinning kan gjøres med en Costas-sløyfe [13, 14], oppkalt etter John P. Costas, som beskrev en metode for å forbedre ytelsen i analoge amplitudemodulerte systemer [15].

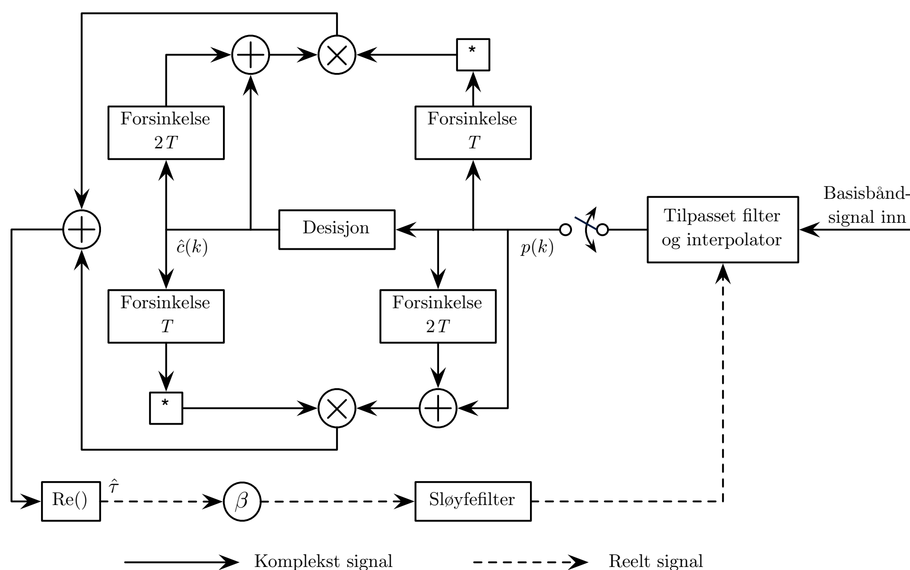
Et BPSK-signal blir generert fra et bærebølgesignal ω_c, som moduleres ved å skifte fasen med 0 eller π ved en bestemt symbolrate. En enkel mikser eller ordinær faselåst sløyfe kan ikke brukes til å gjenvinne den binære informasjonen ettersom det ikke finnes noen spektrallinjekomponent ved $\pm\omega_c$. Som følge av at BPSK-signalet har et spektrum som er symmetrisk rundt den undertrykte bærebølgen, kan Costas-sløyfen brukes til å finne et koherent referansesignal, som vist i figur 3.8.

Virkemåten baserer seg på at den spenningsstyrte oscillatoren (VCO) er låst til den undertrykte bærebølgefrekvensen på inngangen, men med en konstant fasefeil ϕ_e. De to kvadratur-utgangssignalene multipliseres sammen og filtreres med et lavpassfilter (LPF) som har knekkfrekvens nær 0 Hz, slik at filteret fungerer som en integrator som fremskaffer den nødvendige kontrollspenningen, $K \sin 2\phi_e$.

Costas-sløyfen har to stabile punkter 180° fra hverandre, hvilket betyr at Costas har 50 % sjanse for å bomme med 180° når den låser på bærebølgen. For å løse dette kan en kjent pilotsekvens brukes til å detektere polariteten på signalet. Et annet alternativ er å bruke differensiell koding og dekoding, slik at polariteten ikke har betydning for det dekodede signalet.



Figur 3.8: Costas-sløyfe for bærebølgegjennvining.



Figur 3.9: Modifisert Mueller og Müller algoritme for QPSK. Blokkene med en stjerne (*) innebærer komplekskonjugering.

3.7 Taktgjenvinning med mM&M-algoritmen

Taktgjenvinning er viktig for å foreta desisjon på optimale tidspunkt, samt å unngå symbolglipp. Taktgjenvinning og symbolsynkronisering kan utføres ved hjelp av en modifisert Mueller & Müller-algoritme (mM&M) [16]. Denne skiller seg fra den originale M&M-algoritmen ved at den er fri for selvstøy, i tillegg til at simuleringer har vist at det ikke blir noen symbolglipp ved medium til høyt signal-støy-forhold (SNR).

Algoritmen er analysert og dokumentert i [16], og basert på et system med synkroniserte datasymboler med additiv hvit gaussisk støy og perfekt bærebølgesynkronisering. Hensikten med algoritmen er å finne tidsfeilen τ slik at det kan korrigeres for denne. Dette er viktig for å kunne foreta desisjon på det optimale tidspunktet, midt i symbolet. Algoritmen har også lav beregningskompleksitet og opererer bare på én punktprøve per symbol for å finne tidsfeilen. For å finne tidsfeilen trengs det $2N_i + 4$ reelle multiplikasjoner og $2N_i + 5$ reelle addisjoner per tidsfeilestimat $\hat{\tau}$, hvor N_i er antallet filterkoefisienter i det interpolerende FIR-filteret. Prinsippet for mM&M-algoritmen er vist i figur 3.9.

Analyse av algoritmen er utenfor fokuset til denne rapporten, men den generaliserte optimaliserte Mueller og Müller-algoritmen for QPSK er gitt av

$$\mu(k) = \Re\left\{[\hat{c}(k) - \hat{c}(k-2)]\, p^*(k-1) + \hat{c}^*(k-1)\,[p(k) - p(k-2)]\right\}. \quad (3.10)$$

Her er $\hat{c}(k) = \hat{a}(k) + j\hat{b}(k)$ mottakerens beslutning om den reelle og den

imaginære komponenten til dataene. Signalet $p^*(k) = p_r(k) - jq_i(k)$ er det komplekskonjugerte utgangssignalet fra det punktprøvde signaltilpassede filteret. $\Re\{x\}$ er den reelle delen av den komplekse verdien x.

Simuleringer utført med en sløyfefaktor på $\beta = 0{,}18$ og initiell tidsfeil på en halv symbolperiode har vist at mM&M-algoritmen svinger seg inn på 8–10 symbolperioder [16].

3.8 Digital frekvenskonvertering

Fokuset i dette og det neste delkapitlet er på metodene som brukes til digital opp- og nedkonvertering av frekvens og punktprøvingsrate i USRP. Delkapitlene baserer seg i hovedsak på [17, 18].

I en ideell programvaredefinert radio vil analog-digital-omforming foretas direkte i RF-båndet uten bruk av mellomfrekvenser. På grunn av praktiske begrensninger i dagens teknologi, er det nødvendig bruke analoge komponenter til å mikse ned signalet fra RF til en mellomfrekvens før punktprøving og digitalisering i en mottaker. En annen mulighet er å digitalisere signalet på en mellomfrekvens, og gjøre resten av nedkonverteringen digitalt.

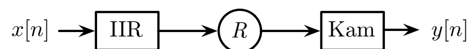
Ulike alternative fremgangsmåter kan brukes når signalet først er hentet ned til mellomfrekvensen. Dette inkluderer direkte nedkonvertering, der det ikke brukes noen egentlig mellomfrekvens, men der RF-signalet konverteres direkte ned til basisbånd. Et annet alternativ er flere analoge miksesteg i en superheterodynmottaker. Denne gjør det lettere å filtrere signalet ved å bruke flere filtre av lav orden i de forskjellige stegene.

Det er nødvendig å generere et lokalt referansesignal for å kunne mikse ned det mottatte signalet. Et alternativ er å bruke en oppslagstabell for komplekse bærebølger, som krever at det lagres mye data. Et annet er å bruke IIR-filtre som oscillatorer, men dette krever multiplikatorer som ikke er tilgjengelig i FPGA-brikken på USRP [18]. «Coordinate Rotation Digital Computer» (CORDIC) er en algoritme for beregning av hyperbolske og trigonometriske funksjoner som kan brukes til å generere et referansesignal med de enkle matematiske operasjonene skifting, addisjon og subtraksjon [6]. Dette gjør algoritmen egnet for maskinvareimplementeringer, og den brukes i USRP.

3.9 Endring av punkprøvingsrate

Punktprøvingsraten kan endres ved hjelp av enten interpolering eller desimering. Dette kan foretas i FPGA uten bruk av multiplikatorer ved hjelp av et «Cascaded Integrator-Comb» (CIC)-filter [18], vist i figur 3.10. Det er et

filter som består av to blokker, hvor den ene er en integrator i form av et IIR-filter og den andre en kam i form av et FIR-filter. Integratoren er et bevegelig gjennomsnittsfilter med en tilbakekoblingskoeffisient, $y[n] = y[n-1] + x[n]$. Kammen er et oddesymmetrisk FIR-filter, $y[n] = x[n] - x[n - RM]$, hvor R er endringen i rate og M er den differensielle forsinkelsen som vanligvis er 1 eller 2. CIC-filtre er langt mindre beregningsintensive enn generelle FIR-filtre [19].



Figur 3.10: CIC-filter.

Halvbåndsfiltre er mye brukt i multiratesignalbehandling, spesielt i forbindelse med interpolering og desimering. Det er et FIR-filter der halvparten av koeffisientene er 0, og hvor senterkoeffisienten er 0,5. Et slikt filter er bare realiserbart når det er sentrert rundt $f_s/4$, og har et odde antall koeffisienter. Fordelen med halvbåndsfilteret fremfor andre filtre er at null-koefisientene reduserer antallet multiplikasjoner og nødvendig minne med 50 %. I desimering brukes halvbåndsfiltre umiddelbart etter CIC-filteret, ettersom CIC-filteret ikke har tilstrekkelig dempning i stoppbåndet.

Kapittel 4

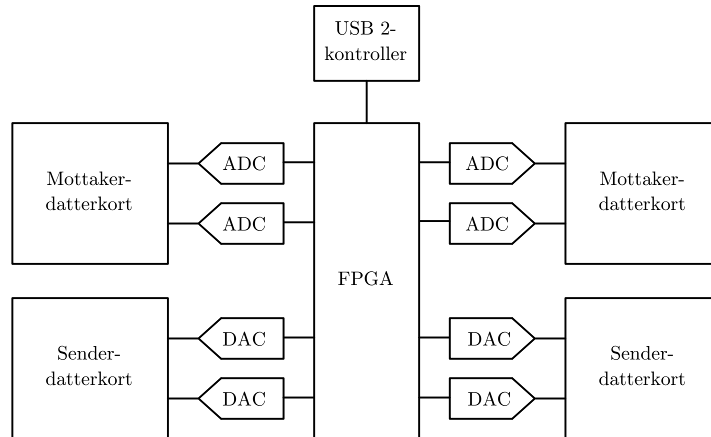
Radioplattformen USRP

Da prosjektet med å utvikle GNU Radio ble startet av Eric Blossom i 1998, var målet å lage en programvarebasert HDTV-mottaker. I den forbindelse hadde han behov for en måte å komme «fra antennen og inn i datamaskinen», og tok kontakt med Matt Ettus. Sistnevnte skaffet til veie forskningsmidler for å utvikle radioplattformen som skulle bli «Universal Software Radio Peripheral» (USRP) [20].

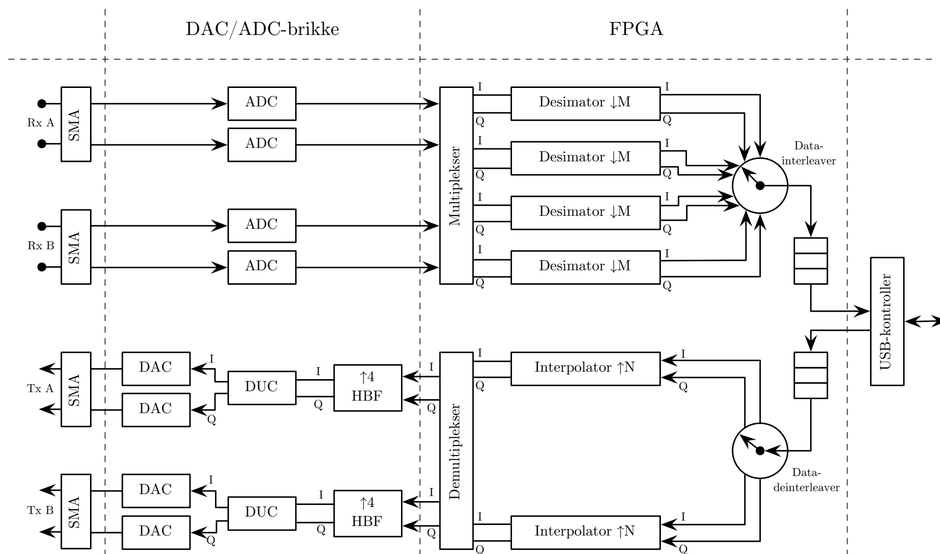
USRP er utviklet for å være en rimelig radioplattform for programvaredefinert radio. Den består av analog-digital- og digital-analog-omformere, en FPGA for høyhastighetssignalbehandling samt en USB 2.0-kontroller som grensesnitt mot en datamaskin. Dette er vist i figur 4.1. Funksjonaliteten til de forskjellige komponentene på hovedkortet til USRPen er vist i figur 4.2. USRP ble utviklet spesielt for GNU Radio, men er delvis støttet av andre SDR-rammeverk, som «Open Source SCA Implementation::Embedded» (OSSIE) [21]. Skjemaer til USRP er fritt tilgjengelig fra nettsiden til GNU Radio-prosektet [22].

En nyere USRP2 med gigabit ethernet-tilkobling i stedet for USB 2.0, og en raskere FPGA, ble tilgjengelig for ordinært salg i mai 2009. Det er blant annet mulig å koble sammen flere USRP2-enheter til komplekse MIMO-systemer [23], men den er ikke ment å skulle erstatte den originale USRP. USRP2 er ikke brukt i denne oppgaven, og resten av kapitlet omhandler den originale USRP-enheten.

USRP har fire 64 MS/s 12-bit analog-digital-omformere (ADC) og fire 128 MS/s 14 bit digital-analog-omformere (DAC). Digital opp- og nedkonvertering av punktprøvingsraten i FPGA er nødvendig for å begrense datamengden som sendes over USB 2.0-grensesnittet. Det brukes en krystalloscillator på 64 MHz som referanse til å generere klokkesignaler i USRPen.



Figur 4.1: Grunnleggende oppbygning av USRP med datterkort.



Figur 4.2: Hovedkortet til USRP med standard FPGA-programvare [22].

4.1 Analog-digital- og digital-analog-omforming

ADC og DAC er implementert på to blandede signalprosessorer (AD9862), hvor hver håndterer et sender- og et mottakerdatterkort. De fire ADC-ene har en presisjon på 12 bit, og en punkprøvingsfrekvens på 64 MS/s. Det dynamiske området er 2 V topp-til-topp. USRP har en programmerbar forsterker før ADC-en for å forsterke svake signaler før digitalisering [17]. Det er også fire DAC-er med presisjon på 14 bit, og en punktprøvingsfrekvens på 128 MS/s. Disse gir ut maksimalt 1 V topp-til-topp med en $50\,\Omega$ differensiell belastning.

4.2 Endring av punktprøvingsrate i FPGA

FPGA er en enhet med programmerbar logikk, og i USRP brukes Altera Cyclone EP1C12 FPGA [17]. Det er mulig å programmere FPGA-en etter eget ønske, men denne seksjonen tar for seg standardkonfigurasjonen. Standardkonfigurasjonen til FPGA-en innebærer hovedoppgavene: grensesnitt til USB-kontrolleren, først inn – først ut (FIFO)-buffere, data-interleaver fra de forskjellige inngangssignalene før mottaker-FIFO-bufferet, data-deinterleaver mellom sender-FIFO-bufferet og oppkonvertering.

Hovedoppgaven til FPGA-brikken er høyhastighetssignalbehandling for å redusere dataratene til noe som lar seg overføre over USB 2.0. Standardkonfigurasjonen til FPGA-en inkluderer to digitale nedkonverterere (DDC) med fire stegs kaskade-integrator-kam (CIC)-filtre og et påfølgende 31 tappers halvbåndsfilter i kaskade for signalforming [17]. Teorien for endring av punktprøvingsrate ble presentert i kapittel 3.9.

Den digitale oppkonverteringen (DUC) som interpolerer signalet før det sendes til DAC-en gjøres delvis i FPGA, og delvis i selve digital-analog-omformeren i AD9862. Denne interpolerer konstant med 4, slik at punktprøvingsraten ut er 128 MS/s.

USRP-hovedkortet utvides med egne RF-kort, og det er plass til totalt to senderkort og to mottakerkort. Hvert kort har mulighet for to kanaler (en kompleks I+Q-kanal), og det er derfor to AD- eller DA-omformere for hvert kort, som vist i figur 4.1 på forrige side. Det finnes også transiverkort som legger beslag på to plasser.

4.3 Båndbreddebegrensninger

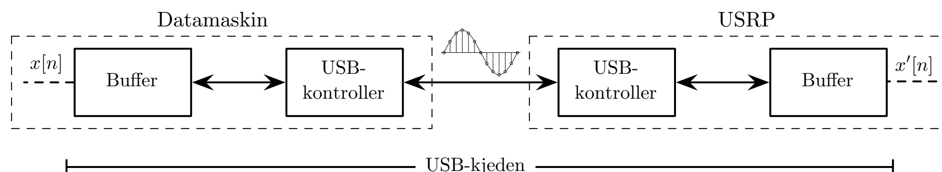
USB 2.0-grensesnittet har en teoretisk overføringsrate på 480 Mbit/s. På grunn av ekstra kompleksitet i USB-protokollen, blir den effektive overføringshastigheten mellom USRP og en datamaskin cirka 256 Mbit/s [17]. Ettersom punktprøvene overføres som to byte på til sammen 16 bit er den maksimale punktprøvingsraten overført over USB

$$f_s \leq \frac{256 \text{ Mbit/s}}{16 \text{ bit/S}} = 16 \text{ MS/s}, \tag{4.1}$$

hvilket betyr at det kan overføres maksimalt 16 megapunktprøver per sekund. Denne begrensningen gjør USRP uegnet til høyhastighetssystemer, ettersom båndbredden blir naturlig begrenset av det lave antallet punktprøver som kan overføres per sekund, jamfør Shannons punktprøvingsteorem [24]. Ved simpleks eller halv dupleks-kommunikasjon vil den maksimale båndbredden følgelig være på 8 MHz. Dersom det brukes både I og Q-kanal vil maksimal båndbredde være 4 MHz.

4.4 Forsinkelse i USB

I en komplett radio kan det være flere kilder til forsinkelse, noe som blant annet kan ha betydning i forbindelse med mediumtilgangskontroll (MAC). I en radio basert på USRP vil det i tillegg komme en betydelig forsinkelse i forbindelse med overføring av punktprøvene via USB, som illustrert i figur 4.3.



Figur 4.3: Overføring av punktprøver mellom en datamaskin og USRP over USB.

Forsinkelsen i USB 2.0 er avhengig av dataraten, ettersom de minste pakkene som kan sendes over USB-protokollen er på 512 byte. Dette innebærer lengre tidsforsinkelse ved lave datarater, ettersom det vil være nødvendig med en lengre mellomlagringstid i et buffer for å sette sammen en USB-pakke. Øvre pakkegrense bestemmes av brukeren via to parametere, antall pakker n og størrelsen på datablokkene z_b. Tiden det tar en punktprøve $x[n]$ å bli overført via USB kan uttrykkes som

$$\Delta_{USB} = \frac{f\left(512, n \cdot z_b\right)}{z_s \cdot f_s}, \tag{4.2}$$

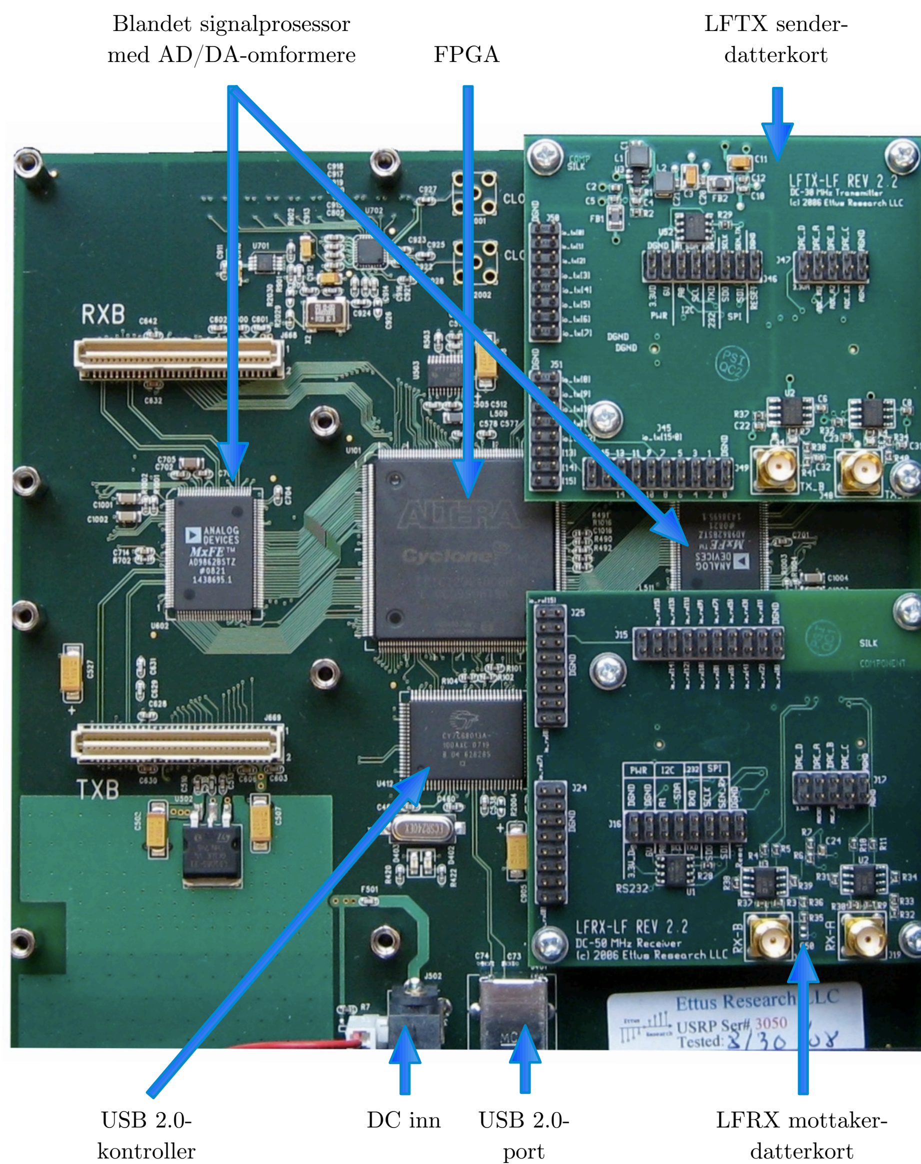
hvor z_s er størrelsen på en punktprøve og f_s er punktprøvingsfrekvensen [25]. Datamengden i bufferet $f(x, y)$, er minst x og maksimalt y.

4.5 Datterkort

RF-grensesnittet til USRP består av separate datterkort, og det eksisterer et utvalg av datterkort for både sending og mottak, som dekker hele frekvensspekteret fra 0 Hz (DC) til 5,9 GHz. Datterkortene finnes som enkle sender- eller mottakerkort, og som mer komplette transivere. De enkleste datterkortene, som LFRX og LFTX, krever en ekstern radioforsats («RF front end»). De mer avanserte kortene kan brukes direkte uten ekstra komponenter dersom det ikke kreves høy uteffekt.

LFRX- og LFTX-datterkortene kan brukes til å sende og motta basisbåndsignaler via en ekstern radioforsats. De har differensielle forsterkere for å kunne håndtere frekvenser ned til 0 Hz. LFRX-kortet har et 30 MHz antialiasingfilter, og begge kortene har en impedans på 50 Ω [17, 26]. Hvert kort har to separate kanaler som kan kombineres til en kompleks kanal (I+Q). USRP-enheten med LFTX- og LFRX-datterkortene som er brukt i denne masteroppgaven, er vist i figur 4.4.

Signaler i 5,8 GHz-området kan behandles uten ekstra RF-utstyr ved å bruke datterkortet XCVR2450. Dette dekker 2.4–2.5 GHz og 4.9–5.9 GHz, som inkluderer ISM-båndet hvor DSRC befinner seg. Dette datterkortet har en maksimal utgangseffekt på 100mW (20 dBm) [27] uten en ekstern forsterker.



Figur 4.4: USRP med LFTX- og LFRX-datterkortene.

Kapittel 5

Frie rammeverk for SDR

Dette kapitlet vil vurdere to frie[1] rammeverk for programvaredefinert radio opp mot hverandre. GNU Radio er et selvstendig rammeverk som ikke bygger på noen standard, mens «Open Source SCA Implementation::Embedded» (OSSIE) er en fri implementering av «Software Communications Architecture» (SCA).

5.1 GNU Radio

GNU Radio er et komplett rammeverk for programvaredefinert radio, og består av en rekke signalbehandlingsblokker og grensesnitt til flere maskinvare- og radioplattformer [29]. GNU Radio kan kjøres på både Linux, Mac OS X og Windows.

Utviklingen av GNU Radio begynte som en avlegger («fork») av Spectrum-Ware/PSpectra, som var en programvareradio utviklet ved Massachusetts Institute of Technology (MIT) på slutten av 1990-tallet.

5.1.1 Arkitektur

GNU Radio bruker en blokkbasert arkitektur som består av signalbehandlingsblokker skrevet i C++ og flytskjemaer skrevet i Python, der flere blokker kobles sammen og utgjør en radioapplikasjon. Et flytskjema i GNU Radio

[1] Fri programvare er programvare med *åpen kildekode*. Dette innebærer at brukeren har lov til å endre og redistribuere programvaren uten spesiell tillatelse. GNU Radio og OSSIE er lisensiert under en mye brukt fri lisens, «GNU General Public License» (GPL) [28].

tilsvarer løst en bølgeform i SCA-sjargong. Denne løsningen medfører at beregningsintensive signalbehandlingsoperasjoner gjøres i C++ for å oppnå høy ytelse, samtidig som Python med sin enkle syntaks og automatiske minnehåndtering gjøre det enkelt å utvikle flytskjemaer. Ettersom signalflyten er definert i et Python-skript kan flytskjemaet endres uten behov for rekompilering.

Python ble oppfunnet i 1991, og anvendes ofte til å binde sammen store programvarekomponenter skrevet i andre programmeringsspråk. Python er et multiparadigmespråk, og det er mulig å programmere prosedyre- eller objektorientert. C++ er et programmeringsspråk fra 1983 med røtter i C [30], men med objektorientering og noe mer moderne funksjonalitet.

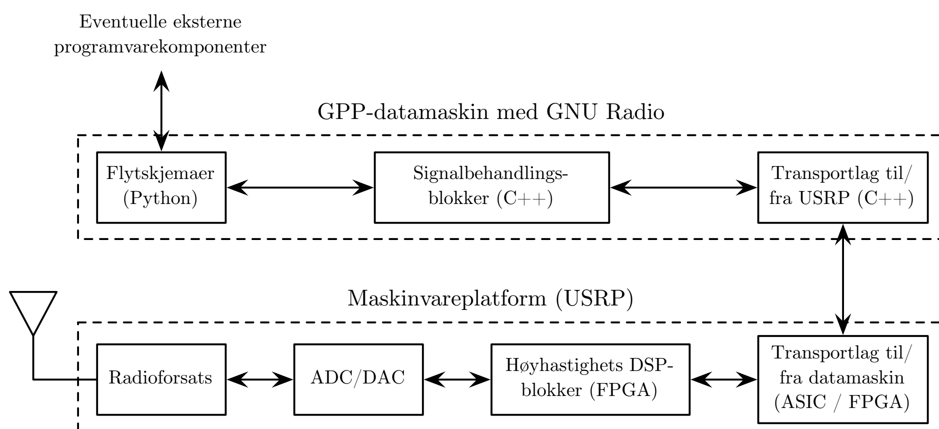
«Simplified Wrapper and Interface Generator» (SWIG) sørger for grensesnittene og kommunikasjon mellom C++ og Python sammen med smarte pekere, som er en del av Boost-biblioteket.

Med de overnevnte teknologiene på plass kan GNU Radio deles inn i to separate deler [18]:

Signalbehandlingsblokker er elementær signalbehandlingsfunksjonalitet som filtre, faseslåste sløyfer, automatisk forsterkningskontroll, og enklere matematiske operatorer der den faktiske signalbehandlingen utføres.

Kjøretidstøttesystemet inkluderer minnebuffere for kommunikasjon mellom blokker, og håndterer dataflyten i flytskjemaet.

Arkitekturen i GNU Radio brukt sammen med USRP er vist i figur 5.1.



Figur 5.1: Et SDR-system med GNU Radio og USRP.

5.1.2 Blokker

Den faktiske signalbehandlingen foregår i signalbehandlingsblokkene, som er skrevet i C++. En slik blokk har et klart definert grensesnitt som består av et antall innganger og utganger med hver sine datatyper, samt en parameterliste.

De fleste blokkene er basert på en av de følgende tre klassene, som alle er subklasser av `gr_block`:

Synkrone blokker er basert på `gr_sync_block`, og har et en-til-en forhold mellom antall elementer på inngangen og utgangen.

Desimerende blokker er basert på `gr_sync_decimator`, og krever $N > 1$ elementer på inngangen for hvert element på utgangen.

Interpolerende blokker er basert på `gr_sync_interpolator`, og produserer $M > 1$ elementer på utgangen for hvert element på inngangen.

De mest sentrale metodene i `gr_block`-klassen, som er nødvendige for å forstå virkemåten til signalbehandlingsblokkene, er:

`general_work()` er metoden som inneholder den faktiske signalbehandlingen, og må overstyres i alle nye blokker. Den kalles automatisk når en blokk aktiveres av «run-time»-systemet.

`gr_make_io_signature()` definerer datatypen og minste og største mulige antall inn- eller utganger.

`set_history()` brukes til å spesifisere hvor mange tidligere elementer som kreves for å generere et enkelt element. Dette gjelder for eksempel FIR-filtre av lengde N. For å generere en punktprøve ut $y[n]$ kreves det tidligere verdier av x fra $x[n]$ til $x[n - M - 1]$.

Hver blokk har også en `forecast()`-funksjon som forteller kjøretidsystemet antall inngangselementer det trenger for å produsere et antall elementer på utgangen, som er knyttet til de overnevnte funksjoner. Når blokken kjøres, forteller den systemet hvor mange elementer på inngangen den har forbrukt.

GNU Radio har flere blokker som kontrollerer USRP-enheten. Punktprøvene som sendes til eller fra USRP-blokken har verdier i området ±32767 (16 bit), som blir skalert til 12 eller 14 bit av USRP-komponenten. Dette er gjort slik for å gjøre flytskjemaer og komponenter uavhengige av den DA/AD-omformeren som brukes i USRP, slik at en endring i maskinvaren ikke skal føre til at bølgeformer eller komponenter slutter å fungere som forventet.

5.1.3 Flytskjemamekanismer

De individuelle blokkene er uavhengige av hverandre. De kan ses på som «svarte bokser» som tar i mot data i den ene enden, behandler dem og sender dem ut i den andre enden, uten noen kjennskap til utenomverdenen. Inngangene og utgangene til en blokk har egne databuffere. Hver inngangsstrøm har et buffer hvor blokken leser data som skal behandles. Etter prosesseringen skriver blokken data til sitt utgangsstrømbuffer. GNU Radio-bufferene er FIFO-buffere som kun er skrivbare for én blokk, men kan leses av et ubegrenset antall andre blokker [31].

Ved kjøring initialiseres flytskjemaet automatisk ved å allokere databuffere tilpasset størrelsen på dataene og lengden på historie til blokken. Videre kobles blokkene sammen, i tråd med det som er spesifisert i Python-koden ved hjelp av `connect`-funksjonen. Når initialiseringen er fullført, overføres kontrollen til fordeleren.

GNU Radio har en fordeler («scheduler») som kjører flytskjemaet. Denne går gjennom alle blokkene i en løkke, helt til alle dataene er blitt konsumert. Fordeleren undersøker hver blokk sekvensielt. Dersom det er en tilstrekkelig mengde data i inngangsbufferet og tilstrekkelig plass i utgangsbufferet, kaller den blokkens `general_work()`-funksjon. Hvis det ikke er tilstrekkelig med data på inngangen, utelates blokken frem til neste runde.

5.1.4 Flytskjemaer

Flytskjemaer programmeres i Python, og oppbygningen tilsvarer tradisjonelle blokkskjemaer. Dette gir en intuitiv forståelse av oppbygningen. Ordinære flytskjemaer defineres som klasser avledet fra klassen `gr.top_block`, og arver dermed alle nødvendige funksjoner for å legge til og koble sammen blokker [32]. Det finnes en rekke metoder for å kontrollere flytskjemaer som gjør det mulig å starte, stoppe, låse og rekonfigurere dem.

Det er mulig å kombinere flere blokker til en ny blokk i Python. Dette baseres på `gr.hier_block2`-klassen, og er spesielt nyttig for å kombinere flere flytskjemaer som skal kjøres samtidig. I GNU Radio er det ikke mulig å kjøre flere blokker basert på `gr.top_block` samtidig. For å lage et duplekssystem, må senderen og mottakeren implementeres for seg som hver sin `gr.hier_block2`-avledede klasse. Disse kobles sammen i én `gr.top_block`-avledet klasse.

Det finnes et grafisk miljø for å lage flytskjemaer, «GNU Radio Companion» (GRC). Det er en integrert del av GNU Radio fra og med versjon 3.2. GRC lar utvikleren koble sammen og konfigurere blokker grafisk, og lagrer sine data i en «Extensible Markup Language» (XML)-fil. Flytskjemaer i Python

genererers automatisk på bakgrunn av XML-filen. GRC fungerer godt til mange anvendelser, men er mindre fleksibelt enn manuell koding i Python.

5.1.5 Pakkebehandling

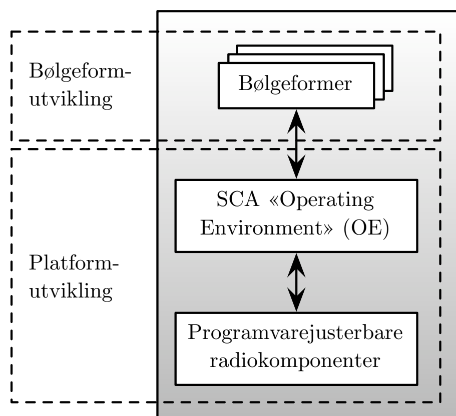
GNU Radio-arkitekturen har tradisjonelt basert seg på kontinuerlige strømmer av data, og har i utgangspunktet ingen begreper om pakker eller meldinger. Dette medfører at det ikke er mulig å lage pakkebaserte systemer utelukkende i GNU Radio. I de eksemplene som finnes, er følgelig mediumtilgangskontroll (MAC) implementert separat. MAC-laget kommuniserer med GNU Radio-flytskjemaet ved å bruke blokker som den eksterne programvaren på MAC-laget kan plassere data i, eller lese data fra. Det finnes flere eksempler på denne fremgangsmåten [25, 31, 33].

For å gjøre det mulig å håndtere datapakker i GNU Radio ble det utviklet en ny arkitektur for flytskjemaer kalt `m-block` («Message Block»). Denne jobber med meldinger istedenfor kontinuerlige strømmer, i motsetning til de tradisjonelle `gr-block`. Foreløpig er det ikke mulig å kombinere `m-block` og `gr-block`, hvilket har medført at `m-block` blir lite brukt. GNU Radio-utviklerene er i gang med å gi alle blokker en mulighet til å forholde seg til enten pakker, kontinuerlige strømmer eller begge deler [34]. Blokker skal da, i tillegg til, eller istedenfor den tradisjonelle `work()`-metoden for å håndtere strømmer, ha en `handle_msg()`-metode for å håndtere pakker [35].

5.2 SCA og OSSIE

For å sikre fleksible og rimelige radioer startet det amerikanske forsvaret på begynnelsen av 1990-tallet utvikling av en plattform for programvaredefinert radio, «Joint Tactical Radio System» (JTRS) [36]. «Software Communications Architecture» (SCA) ble utviklet som et ledd i denne prosessen, og er en programvarearkitektur som bestemmer hvordan komponenter i programvare og maskinvare i en programvaredefinert radio skal fungere sammen. I SCA-spesifikasjonene er det definert fire overordnede mål [37]. SCA skal

- sørge for portabilitet på applikasjonsnivå mellom forskjellige SCA-implementeringer,
- fremtvinge kommersielle standarder for å redusere utviklingskostnader,
- redusere utviklingstiden for programvaren gjennom mulighet for gjenbrukbare moduler, og
- bygge på kommende kommersielle rammeverk og arkitekturer.



Figur 5.2: Prinsipiell lagdeling i JTRS SCA.

For å oppnå dette er det lagt vekt på å møte kommersielle og sivile krav, i tillegg til de rent militære.

SCA inneholder et sett med regler for kommunikasjon og samhandling mellom programvarekomponentene (bølgeformene) og den aktuelle maskinvaren. Disse reglene definerer «Operating Environment» (OE), som består av «Core Framework» (CF), «Common Object Request Broker Architecture» (CORBA) og operativsystemet (POSIX) [38].

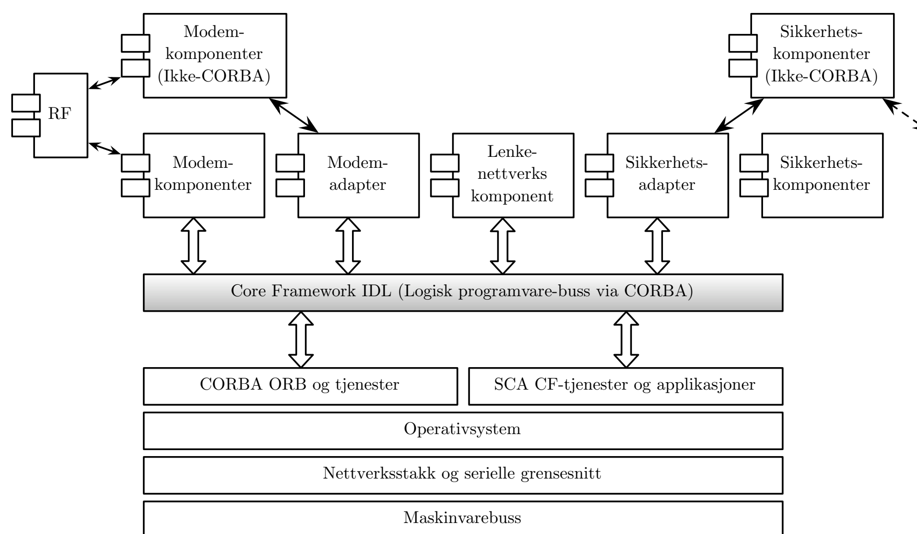
En SCA-basert radio kan deles inn i tre nivåer, hvor SCA OE er limet som binder bølgeformene til maskinvaren og radiokomponentene, som vist i figur 5.2.

De grunnleggende byggeblokkene på bølgeformnivå i SCA er komponenter, noder og enheter. En bølgeformutvikler deler koden opp i komponenter, som typisk kan sammenlignes med blokker i et blokkskjema, eller flytskjemaer i GNU Radio. En node er en fysisk maskinvare, og denne noden kan bestå av en flere enheter. Ofte vil hele maskinvareplattformen være én node [38]. Hver komponent kjøres på en bestemt enhet. Alle OSSIE-komponenter kan ses på som de har to helt separate deler, der den ene er signalbehandlingsfunksjonaliteten, og den andre håndterer SCA-infrastrukturen.

5.2.1 Strukturen i SCA

SCA skiller mellom sikre og usikre komponenter. I praksis gir dette to parallelle og adskilte OE-sett, men for enkelhets skyld viser figur 5.3 bare den sikre delen, som grenser til RF-komponentene [38].

Hovedblokkene i SCA OE er «Core Framework» (CF), CORBA og operativsystemet. OE spesifiserer grensesnitt, regler og prosedyrer som må følges



Figur 5.3: Detaljert fremstilling av SCA OE.

for å implementere en SCA-kompatibel radio [38]. Med disse elementene på plass, skal det være enkelt å flytte en bølgeform fra en type radio til en annen. Resten av dette kapitlet vil gå mer i dybden på funksjonen til hver av de tre hovedblokkene.

I programvaredefinerte radioer kan det være flere typer prosesseringsenheter som foretar signalbehandling og andre beregninger. «Common Object Request Broker Architecture» (CORBA) [39] er en mellomvare som gjør det mulig å kjøre forskjellige prosesser på forskjellige enheter, men likevel la prosessene kommunisere sømløst med hverandre.

I SCA foregår kommunikasjon mellom komponenter via såkalte porter. En komponent som sender fra seg data for at de skal behandles av en annen komponent, opptrer som en klient som benytter tjenester denne tilbyr. Derfor betegnes utgangsporter som «uses»-porter. Tilsvarende er betegnelsen på inngangsporter «provides».

Grensesnittet mellom komponenter i SCA er beskrevet i «Interface Description Language» (IDL). Dette er et språk laget spesifikt for å beskrive hva et objekt tilbyr andre, og grensesnittet det kommuniserer gjennom. Dette gjør det mulig for objekter og komponenter som er implementert med forskjellige programmeringsspråk å snakke med hverandre [40].

Rammeverket «Core Framework» (CF) knytter sammen maskinvare og programvare i radiosystemene. Dette inkluderer maskinvareabstraksjon, og gjør det lett å flytte bølgeformer mellom uavhengige radioapparater der CF er implementert. En komponent og bølgeformutvikler trenger følgelig hovedsakelig bare å forholde seg til CF.

5.2.2 OSSIE

«Open Source SCA Implementation::Embedded» (OSSIE) er en fri implementering av SCA CF, utviklet ved Virginia Polytechnic Institute and State University (Virginia Tech) [41, 42]. OSSIE inkluderer programvare for utvikling og avlusing av komponenter og bølgeformer. OSSIE benytter andre frie programmer, hvor det viktigste er CORBA-implementeringen omniORB [43]. Signalbehandlingsblokkene i OSSIE er skrevet i C++. I motsetning til GNU Radio, har OSSIE fordelen av en pakkebasert arkitektur. OSSIE utvikles for Linux, men det arbeides med å gjøre det lettere å installere det på Mac OS X [42].

OSSIE inneholder grafiske utviklingsverktøy som gjør det mulig å utvikle SCA-kompatible bølgeformer og komponenter uten dyptgående kjennskap til hverken SCA eller CORBA. Dette inkluderer et tillegg til det integrerte utviklingsmiljøet Eclipse, som gjør at OSSIE kan tilby et komplett integrert utviklingsmiljø. Eclipse med «OSSIE Eclipse Feature» (OEF) kan generere skjellettkode til komponenter automatisk, på bakgrunn av informasjon om porter og egenskaper som spesifiseres av utvikleren. I tillegg kan komponenter grafisk kobles sammen til bølgeformer, og tilordnes ulike prosesseringsenheter.

OSSIE fungerer med USRP og flere av datterkortene ved hjelp av enkelte elementer fra GNU Radio, men det er problemer med noen datterkort. Komponenten som brukes til å styre USRP fra OSSIE er også relativt begrenset, noe som gjør OSSIE mindre fleksibelt sammen med USRP enn GNU Radio. Blant annet er det ikke mulig å adressere flere USRP-enheter tilkoblet en maskin, og det er heller ikke mulig å adressere spesifikke datterkort på en enkelt USRP.

5.3 Sammenligning av GNU Radio og OSSIE

GNU Radio har en gammel arkitektur, men kommer ferdig med en rekke blokker, hvilket kan redusere utviklingstiden for nye radioapplikasjoner betraktelig. Det brukes av mange, og fungerer godt med USRP som er laget spesielt for GNU Radio. Den største standardiseringsinnsatsen innen SDR har vært gjort i forbindelse med SCA, og OSSIE er en fri implementering av denne standarden som har fått en betydelig utbredelse. Det eksisterer mange kommersielle produkter relatert til SCA [44].

Et moment som skiller OSSIE vesentlig fra GNU Radio er at det ser ut til å ha langt mindre utbredelse. Det finnes også dramatisk færre ferdige komponenter for OSSIE enn det finnes blokker for GNU Radio. Videre er det langt vanskeligere å oppdrive eksempler på praktiske anvendelser av OSSIE.

Hovedargumentene for å velge GNU Radio er at det fungerer på alle de store operativsystemene, har en stor brukermasse og et godt utvalg i ferdige blokker. Dette inkluderer alt fra grafiske spektrumsanalysatorer og oscilloskop, til modulatorer/demodulatorer og synkroniseringsblokker. Ikke minst har GNU Radio utmerket kompatibilitet med USRP. Ulempen med GNU Radio er at det har en gammeldags arkitektur, der hele flytskjemaet kjører som én prosess. Dette innebærer at det ikke er mulig å kjøre ulike blokker på forskjellige prosesseringsenheter, slik det kan gjøres i OSSIE. Det kan også være en ulempe i seg selv at GNU Radio ikke bygger på SCA.

Det viktigste argumentet for å bruke OSSIE fremfor GNU Radio er at det er basert på SCA, og gjør det mulig å flytte hele eller deler av bølgeformen til en annen plattform dersom det blir aktuelt [45]. SCA har også en arkitektur som gjør det mulig å fordele komponenter ut over flere prosesseringsnoder, og oppfører seg slik at hver komponent er en egen prosess. Ulempen med OSSIE i forhold til GNU Radio er at det er mindre utbredt og at det finnes langt færre ferdige komponenter som kan brukes direkte av bølgeformutviklere. OSSIE har også dårligere støtte for USRP enn GNU Radio.

De grafiske utviklingsverktøyene i OSSIE og GNU Radio skiller seg vesentlig fra hverandre. For en ren bølgeformutvikler vil sistnevnte med GRC være et godt utviklingsmiljø, ettersom GRC har en meget lav brukerterskel. Det er også en fordel for GNU Radio at det kan oppnås stor fleksibilitet, ettersom bølgeformene er definert i Python-kode som er lett å endre. I OSSIE derimot brukes det brukes en omfattende XML-notasjon som ikke er ment å skulle endres manuelt. OSSIE har et godt verktøy for å automatisk generere skjellettkode ved opprettelse av nye komponenter (OEF), noe som må gjøres manuelt i GNU Radio. Dette gjør at terskelen for å lære å lage nye komponenter er noe lavere med OSSIE.

Læringskurven for GNU Radio og OSSIE kan sammenlignes, men det store utvalget ferdige blokker i GNU Radio gjør det mulig å realisere en rekke ulike radiosystemer uten å lage egne komponenter. Dette innebærer at brukerterskelen for å implementere nyttige bølgeformer er lav med GNU Radio. For å bruke OSSIE er det nødvendig å lage egne komponenter fra begynnelsen, men det er lettere å komme i gang med å lage komponenter i OSSIE takket være muligheten til å generere skallkode i OEF. Blokker i GNU Radio må skrives fra bunnen av, og det er en rekke filer som må redigeres, noe som kan være mindre oversiktlig for nybegynnere.

5.3.1 Kompleksitet og ytelse

På SDR Forum sin årlige konferanse i 2008 ble det presentert to artikler som sammenligner GNU Radio og OSSIE [44, 46]. Disse analyserte ytelses-

forskjellene, men kom frem til ulike konklusjoner, noe som kan tyde på at hvilket av de to rammeverkene som yter best er avhengig av hvordan de brukes.

Et enkelt full dupleks-system med BPSK-modulasjon ble implementert med begge rammeverkene i [46]. Dette kjørte på en maskin uten radioutstyr, og ikke over luften, slik at systemet behandlet data så fort som mulig på den aktuelle maskinvaren uten noen form for tidssynkronisering. Ytelsen ble målt ved å se på den maksimale overføringsraten. Med like store pakker fant denne undersøkelsen at den maksimale overføringsraten var 0,72 Mbit/s med OSSIE og 0,59 Mbit/s med GNU Radio. Ifølge [46] var dette overraskende lite. Den undersøkelsen konkluderte med at med standardkomponenter ser OSSIE ut til å være marginalt raskere og ha noe mindre beregningskompleksitet enn GNU Radio.

I [44] sin samtidige undersøkelse ble beregningskompleksiteten vurdert ved hjelp av profileringsverktøy som analyserer minnebruk og prosessorbelastning. Denne undersøkelsen fant at for OSSIE-komponenter med enkel signalbehandling eller små pakkestørrelser, var andelen av prosessortid som gikk med til andre ting enn signalbehandling (SCA-infrastrukturen og lignende), på over 70 %. Dette viser viktigheten av å ha en betydelig mengde signalbehandling i hver blokk for å begrense effekten av SCA-infrastrukturen. Ved lik oppdeling i blokker/komponenter konkluderer artikkelen med at for den samme bølgeformen kreves det fem ganger så mye prosessorkapasitet, og to ganger så mye minne (RAM) med OSSIE som med GNU Radio.

5.3.2 Interkomponentlatens

Det er store forskjeller i latensen mellom komponentene i de to rammeverkene på grunn av den totalt forskjellige arkitekturen. I GNU Radio er interkomponentkommunikasjonen en ren parameteroverføring, mens det i OSSIE er en komplett protokoll for sending og overføring av data via CORBA, med alt det gir av fleksibilitet og økt kompleksitet.

Ved å implementere en FM-mottaker med begge rammeverkene har [44] vist at interkomponentlatensen er betydelig større i OSSIE enn i GNU Radio. For store pakker (rundt 4096 punktprøver per pakke) er latensen 25 ganger større i OSSIE enn i GNU Radio, 146 μs mot 5,7 μs[2]. Artikkelen konkluderer med at forsinkelsene som følge av CORBA gjør en dupleks bølgeform som GSM urealiserbar med OSSIE.

[2]Den eksakte latensen avhenger av maskinvaren som brukes.

5.3.3 Oppsummering

Ut fra artiklene og betraktningene i de foregående to delkapitlene, er det vanskelig å slå entydig fast hvilket system som krever minst maskinvareresurser. Det er likevel interessant at OSSIE har betydelig større latens i dataoverføring mellom komponenter enn GNU Radio.

De viktigste forskjellene, fordelene og ulempene til GNU Radio og OSSIE er oppsummert i tabell 5.1.

Tabell 5.1: Sammenligning mellom GNU Radio og OSSIE.

	GNU Radio	OSSIE
USRP-støtte	ja	delvis
USRP2-støtte	ja (kun på Linux foreløpig)	nei
Fri programvare	ja	ja
Plattformer	Linux, Mac OS X, Windows	Linux
Utbredelse	stor	liten
SCA-kompatibel	nei	ja
Pakkebasert	nei (delvis)	ja
Utvalg av blokker/komponenter	stort	lite
Interkomponentlatens	5,7 μs	146 μs
Beregningskompleksitet	høy	høy

Kapittel 6

Dedikert kortholdslink

Dedikert kortholdslink (DSRC) er et system for veitransporttelematikk. Dette kapitlet gir en grunnleggende oversikt over DSRC-standardene med et spesielt fokus på fysisk lag. Hovedkilden for dette kapitlet er [47].

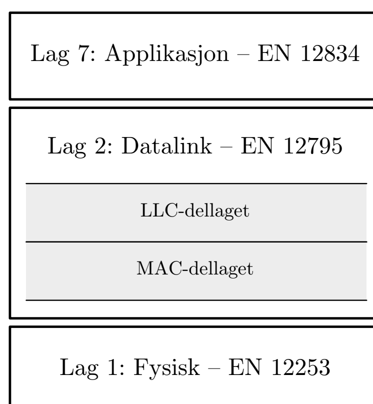
Det eksisterer flere teknologier som er egnet for ulike sider ved veitransporttelematikk, blant annet mobiltelefonnett for store dekningsområder, og DSRC for lokal kommunikasjon innenfor begrensede kommunikasjonssoner langs veinettet. Det eksisterer forskjellige DSRC-løsninger i ulike regioner i verden, og i Europa og Nord-Amerika brukes ulike systemer.

I Norge brukes DSRC først og fremst til bompengebetaling med AutoPASS-systemet, men er i ferd med å få nye anvendelsesområder. Veidirektoratet har nylig foreslått andre anvendelsesområder, som betaling av drivstoff, bilvask og parkering ved hjelp av AutoPASS-brikken [48].

DSRC er et lagdelt kommunikasjonssystem, med en protokollstakk som er noe forenklet i forhold til OSI-modellen [49], som vist i figur 6.1. Modellen inneholder de to laveste lagene i den tradisjonelle OSI-modellen, samt applikasjonslaget på toppen.

Den europeiske standardiseringsorganisasjonen CEN har utgitt fire standarder for DSRC. Disse er fastsatt som norsk standard (NS). De norske standardene er engelskspråklige, og skiller seg bare fra de opprinnelige standardene med prefikset NS i standardnavnet, og en norsk tittel:

- NS-EN 12253 Veitransporttelematikk — Meldinger via dedikert kortholdslink — Fysisk lag ved bruk av 5,8 GHz mikrobølge [50]
- NS-EN 12795 Veitransporttelematikk — Dedikert kortholdslink (DSRC) — DSRC-datakjedelag: Mediumtilgang og logisk linkkontroll [51]



Figur 6.1: Protokollstakken i DSRC.

- NS-EN 12834 Veitransporttelematikk — Dedikert kortholdslink (DSRC) — DSRC-anvendelseslag [52]
- NS-EN 13372 Veitransporttelematikk (RTTT) — Meldinger via dedikert kortholdslink — Profiler for RTTT-anvendelser [53]

I tillegg kommer ISO/TS 14907-1 [54], som er en standard som bestemmer hvordan radioutstyret skal testes for å verifisere at det er innenfor de gitte spesifikasjonene.

Det eksisterer tre ulike systemer i 5,8 GHz-båndet som refereres til som LDR (lav datarate), MDR (medium datarate) og HDR (høy datarate). MDR brukes i Norge og er mest utbredt i Europa. Resten av dette kapitlet vil gi en oversikt over de mest sentrale delene av standardene listet opp over, med hovedfokus på fysisk lag ved medium datarate.

6.1 Fysisk lag ved 5,8 GHz

Det fysiske laget i DSRC ved 5,8 GHz er bestemt av EN 12253 [50]. Standarden definerer frekvensområder, frekvensfeiltoleranse, krav til demping av sidelober, og modulasjonsmetoder. DSRC bruker et bånd på 10 MHz midt i det lisensfrie ISM-båndet, ved 5,795–5,805 GHz. I enkelte land brukes også et ekstra bånd i området 5,805–5,815 GHz. Hvert av disse båndene er delt opp i to delbånd á 5 GHz, og det er derfor totalt fire mulige kanaler i DSRC.

Det benyttes ikke feilkorrigerende koding i DSRC. Dette har sammenheng med de korte rammene, og en bitfeilrate i området BER $= 10^{-6}$. Ved maksimal rammelengde på 1096 bit er sannsynligheten for feil i en ramme på $1096 \cdot 10^{-6} = 1{,}096 \cdot 10^{-3}$, omkring én til tusen.

Figur 6.2: Rammestruktur for fysisk lag i DSRC.

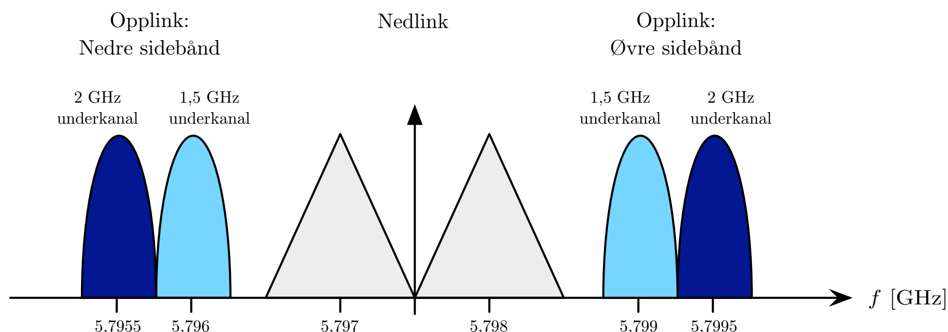
6.1.1 Nedlink

Nedlinkparametrene bestemmer transmisjon fra veikantenheten (RSU) til bilbrikken (OBU). Det er definert tre forskjellige spektrummasker: Klasse A, B og C. Klasse C har de strengeste kravene, og klasse A skal ikke brukes i nye installasjoner. Videre stilles det krav til maksimal effektiv isotropisk utstrålt effekt (EIRP) og polarisasjon.

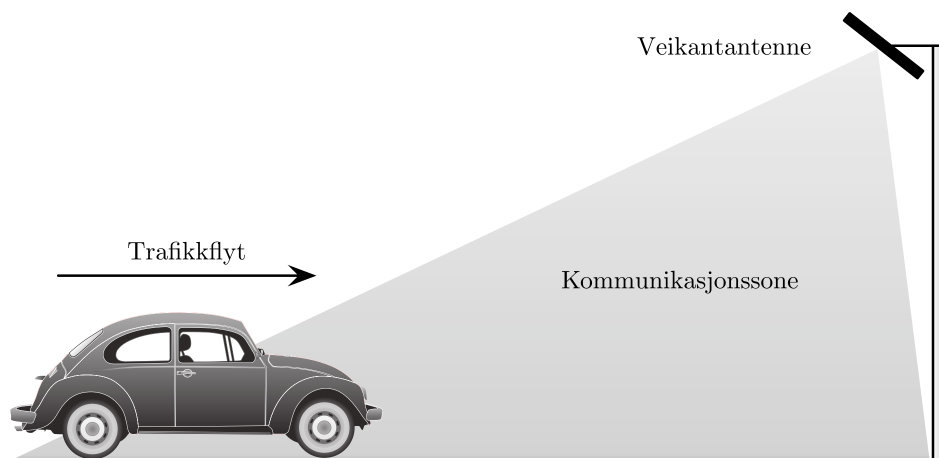
Det skal brukes en tonivå-amplitudeskiftnøkling med en modulasjonsindeks på 0,5–0,9. Dataene FM0-kodes ved at symbolet for binær 1 kun har overgang ved begynnelsen og slutten av bitintervallet. Symbolet for binær 0 har i tillegg en overgang i midten av intervallet. Bitraten er 500 kbit/s, og toleransen for klokkefeil er på ± 100 ppm. Amplitudemodulasjon og linjekoding er behandlet i kapittel 3.3 og 3.4. Det brukes et preambel på 16 bit ± 1 bit som består av en sekvens av enere. Se figur 6.2. Bitfeilraten skal være mindre enn 10^{-6} når mottatt effekt er innenfor kravene.

6.1.2 Opplink

Kommunikasjonen fra bilbrikken til veikantenheten foregår på en underkanalfrekvens («subcarrier frequency») av 1,5 eller 2,0 MHz. Det kreves at bilbrikken støter begge, men veikantenheten kan støtte bare en, eller begge samtidig. Figur 6.3 viser kanalorganiseringen i spekteret med en nedlink, og to opplink-kanaler forskjøvet med 1,5 eller 2 MHz og medium datarate (MDR).



Figur 6.3: Frekvensspektrum for den laveste DSRC-kanalen.



Figur 6.4: Kommunikasjonssonen er i hovedsak foran antennen.

Underkanalfrekvensen har en feiltoleranse på ±0,1 %. Med 0,1 % feil, og en 2,0 MHz underkanal, gir dette en frekvensfeil på

$$f_e = \pm 0{,}001 \cdot 2{,}0\,\mathrm{MHz} = 2000\,\mathrm{Hz}. \quad (6.1)$$

Dataraten på opplinkkommunikasjonen er 250 kbit/s. Modulasjonsmetoden er binær faseskiftnøkling, som ble introdusert i kapittel 3.1 på side 6. Det brukes en differensiell linjekode, NRZI, hvilket innebærer at modulasjonen kan ses på som differensiell BPSK, som ble beskrevet i kapittel 3.2. Toleransen for klokkefeil er ±1000 ppm. I DSRC endres fasen ved binær 0, mens den holdes uendret for binær 1. Det kreves at det brukes et preambel av lengde 32–40 μs modulert med NRZI-kodede 1 bit (ren bærebølge), tilsvarende 8–10 symboler, etterfulgt av 8 NRZI-kodede 0 bit.

Kommunikasjonssonen er definert som det romlige området hvor OBU er plassert slik at et utsendt signal mottas av RSU med en bitfeilrate mindre enn 10^{-6}. Kommunikasjonssonen er i hovedsak foran antennen, mot bilen, som vist i figur 6.4.

6.2 Mediumtilgang og logisk linkkontroll

Mediumtilgang og logisk linkkontroll er på lag 2 i OSI-modellen, og bestemmes av EN 12795 [51]. Den støtter både kringkasting og halv duplekstransmisjon, og tar høyde for at det mobile utstyret passerer gjennom en begrenset sone hvor kommunikasjon er mulig.

Standarden skiller klart mellom mediumtilgangskontroll (MAC) og logisk

linkkontroll (LLC), og definerer MAC-prosedyrer, adresseringsregler, flytkontrollregler, feilkontroll og primitiver mot applikasjonslaget.

6.2.1 Rammestruktur

Rammestrukturen med data er vist i figur 6.5. Rammer uten datafelt (LPDU) brukes til forespørsler om mediumaksess og allokering.

Start-flag (8 bit)	Link-adressefelt (8–32 bit)	MAC-kontrollfelt (8 bit)	LPDU (N·8 bit)	Rammekontroll-sekvens (16 bit)	Stopp-flag (8 bit)

Figur 6.5: Rammestrukturen i DSRC.

Alle rammer starter og slutter med et flagg. Flagget er en fast sekvens av åtte bit (01111110). I mottaksmodus skal alle stasjoner lytte kontinuerlig etter denne sekvensen, og sendere skal kun sende komplette åtte-bit flagg. Det brukes innsetting av en 0-bit for å forhindre at en flaggsekvens tilfeldigvis kommer i andre felt i rammen.

Mottakeren skal tolke ett enkelt flagg som et stopp- *og* startflagg. Uten dette ville feilaktig detekterte rammer som følge av støy kunne blokkere for virkelige rammer. Dersom mottakeren detekterer seks enere skal det syvende bitet inspiseres. Dersom det er 0 er det blitt detektert et flagg, og dersom det er 1 er hele rammen ugyldig. Dette innebærer at det ikke er nødvendig på fysisk lag å vurdere om de mottatte dataene er reelle, eller som følge av støy.

Linkadressefeltet inneholder linkidentifikatoren (LID). Denne er enten en privat LID på fire oktetter, eller en multicast- eller kringkastings-LID på en oktett. Datafeltet LPDU inneholder et LLC-kontrollfelt, og $N \cdot 8$ bit med data. Den maksimale verdien av N er gitt av parametrene $N2 = 128$, $N3 = 128$ eller $N4 = 9$ i standarden [51]. Dette er maksimalt antall oktetter for henholdsvis nedlink, privat opplink og offentlig opplink. Rammekontrollsekvensen er en 16 bit sekvens som brukes til deteksjon av feil i de øvrige feltene i rammen.

Med parametrene ovenfor vil den maksimale rammestørrelsen være på totalt 1096 bit. Ved 500 kbit/s tilsvarer dette en tid på

$$t = \frac{1096\,\text{bit}}{500\,\text{kbit/s}} = 2{,}192\,\text{ms}, \tag{6.2}$$

og ved 250 kbit/s er tiden det dobbelte,

$$t = \frac{1096\,\text{bit}}{250\,\text{kbit/s}} = 4{,}384\,\text{ms}. \tag{6.3}$$

Bitrekkefølgen ved overføring er slik at flagg, linkadresse, MAC-kontrollfelt og LPDU sendes med det minst signifikante bitet (LSB) først i hver oktett. I Rammekontrollsekvensen sender koeffisienten med høyest verdi først.

6.2.2 Mediumtilgangskontroll

Mediumtilgangskontroll og logisk linkkontroll er delt opp i to separate dellag. Ingen av lagene blir dekket i detalj i dette kapitlet ettersom de har begrenset relevans for dette prosjektet.

Mediumtilgangskontroll (MAC) bruker halv dupleks og asynkron tidsdelt multippel aksess (TDMA). Veikantenheten kontrollerer det fysiske mediet, og må gi tillatelse til mobilenheten før den kan sende, men mobilenhetene kan be om tilgang til mediet ved hjelp av «random» aksess. Kommunikasjon initialiseres alltid fra veikantenheten, som starter kommunikasjonen ved å sende informasjon til mobilenheten i form av en «Beacon Service Table» (BST)-ramme.

Opplinkvinduer allokeres av veikantenheten, og indikeres til bilbrikken via MAC-kontrollfeltet på nedlinkrammer. Opplinkvinduet følger da umiddelbart etter i tid. Det skilles mellom private opplinkvinduer der det gis tilgang til en bestemt bruker, og vinduer som kan brukes til tilfeldig aksess.

For å unngå overflødige data i overføringen forgår all kommunikasjon i en tilkoblingsfri modus. LLC-dellaget støtter likevel at pakker kan sendes både med og uten bekreftelse. Ubekreftede tjenester brukes både på opp- og nedlink. Bekreftede tjenester kan brukes for dataoverføring til én OBU.

6.3 Applikasjonslaget

Applikasjonslaget er dekket av standarden EN 12834 [52], som sammen med EN 13372 (Profiler for RTTT-anvendelser) [53] bestemmer anvendelsesområder for DSRC innen veitransporttelematikk. Et eksempel på en slik anvendelse er AutoPASS-systemet for automatisk innkreving av bompenger.

Kapittel 7

Realisering av en veikantenhet

En enkel DSRC RSU-sender og en koherent mottaker for medium datarate (MDR) ble implementert ved hjelp av GNU Radio og USRP med LFRX- og LFTX-datterkortene, samt en radioforsats fra Q-Free. Det ble også arbeidet med en sender utviklet med OSSIE og en ikke-koherent mottaker med GNU Radio. Dette kapitlet presenterer valgene som ble tatt, og hvordan systemene ble testet.

7.1 Valg av rammeverk

En del av oppgaven er å vurdere hvilket av de to rammeverkene som er best egnet til å implementere en prototype av en veikantenhet for dedikert kortholdslink (DSRC). Dette må ses i sammenheng med dedikert kortholdslink som presenteres i kapittel 6, og sammenlikningen mellom GNU Radio og OSSIE i kapittel 5.3.

Systemet som skal realiseres overfører maksimalt 500 kbit/s, hvilket er mulig med begge de aktuelle rammeverkene. Brukerbasen og utvalget av ferdige blokker taler til GNU Radios fordel, og det samme gjør den gode støtten for USRP, som skal brukes i realiseringen. Den store interkomponentlatensen diskvalifiserer OSSIE fra praktiske anvendelser, ettersom en interkomponentlatens på 146 μs er for lang for datalinklaget og MAC-dellaget i DSRC [47, s. 39]. Dette veier opp for problemene med dårlig støtte for pakkekommunikasjon i GNU Radio, spesielt ettersom det arbeides med å gjøre støtten for pakker bedre i nær fremtid. Høyere lags protokoller kan også implementeres uavhengig av GNU Radio.

På bakgrunn av dette ble det besluttet å fokusere på en realisering med

GNU Radio, men også lage en realisering av senderen i OSSIE for å kunne se systemene opp mot hverandre i praktisk bruk. Det er blitt implementert en sender og en koherent mottaker i GNU Radio. Det ble også jobbet med en ikke-koherent mottaker. Videre ble det laget en sender i OSSIE, som bygger på senderen fra GNU Radio. Realiseringene behandles for seg i de påfølgende delkapitlene.

7.2 Utstyr

GNU Radio ble installert og kjørt på Mac OS X, og OSSIE ble kjørt på Linux-distribusjonen Ubuntu 8.04 gjennom virtualiseringsplattformen VMware Fusion. Tabell 7.1 viser det viktigste radio- og datautstyret som ble brukt i prosjektet. Programvaren som ble brukt er listet opp i tabell 7.2.

Tabell 7.1: Maskinvare som er benyttet i den praktiske realiseringen.

Type	Versjon/spesifikasjoner
USRP-hovedkort	4.5
LFRX-datterkot	2.2
LFTX-datterkort	2.2
Radioforsats fra Q-Free	—
Bærbar datamaskin	2,4 GHz Intel Core 2 Duo CPU, 4 GB RAM

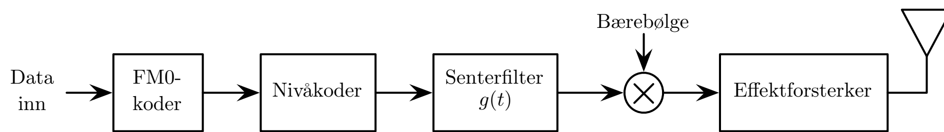
Tabell 7.2: Programvare som er benyttet under arbeidet med den praktiske realiseringen.

Navn	Versjon
Mac OS	10.5.7
Ubuntu	8.04
VMware Fusion	2.0.4
GNU Radio	3.2
OSSIE	7.2

7.3 Sender

Senderen i veikantutstyret bruker amplitudeskiftnøkling, som er behandlet i kapittel 3.3 på side 8. En generell DSRC RSU-sender vil være basert på prinsippene i figur 7.1. Resten av dette delkapitlet vil ta for seg hvordan denne senderen er realisert.

Sammen med USRP med LFTX-datterkortet brukes det en radioforsats som er utviklet av Q-Free. Den tar inn signalet som skal sendes som et basisbånd-

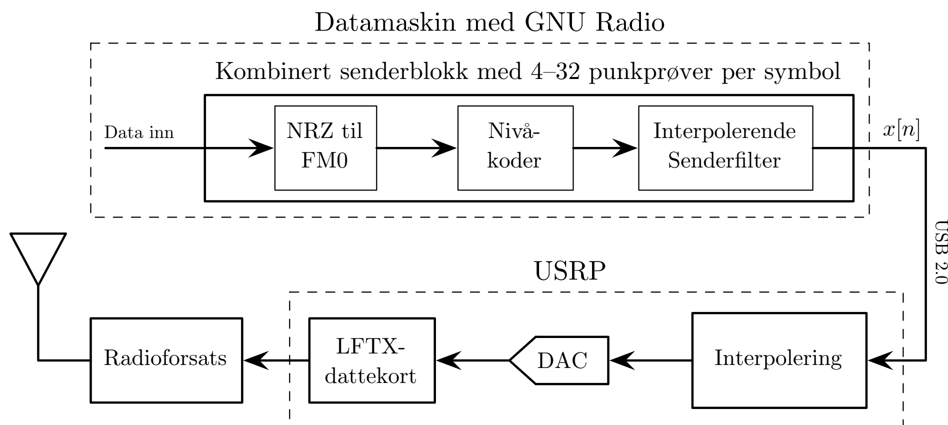


Figur 7.1: Generell DSRC RSU-sender.

signal (tilkoblet via en SMA-kontakt), og mikser signalet opp til 5,8 GHz og forsterker det til ønsket uteffekt. Radioforsatsen er «dum», og kan ikke styres fra programvare. Den sender kontinuerlig ut en bærebølge på 5,8 GHz som trengs for å gi bilutstyret effekt, også når det ikke sendes data.

7.3.1 Sender implementert med GNU Radio

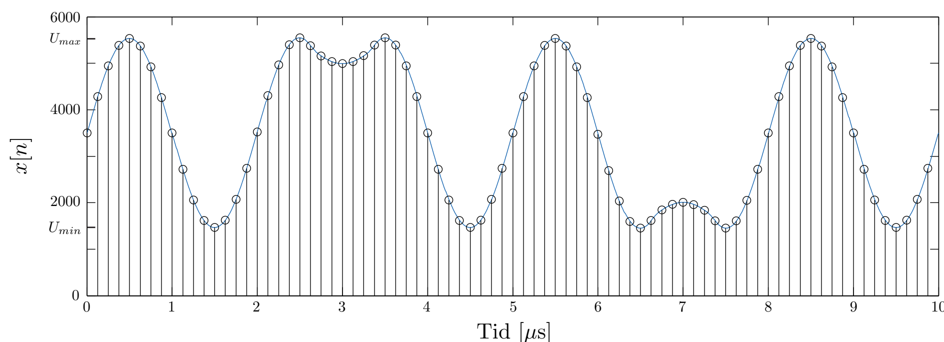
Senderen ble implementert som en kombinert senderblokk bestående av tre delblokker (`dsrc_pulse_shaper_bs`), som vist i figur 7.2. Ettersom senderen bruker en enkel tonivå amplitudemodulasjon, består hele signalveien av reelle signaler. Det brukes en USRP-kontrollblokk som ikke er vist i figuren, men som kontrollerer funksjonaliteten i USRP.



Figur 7.2: Senderen slik den er realisert med GNU Radio og USRP. Basisbåndsignalet $x[n]$ er gjengitt i figur 7.3 på neste side.

For å tilfredsstille kravene til spektralmasker som settes av standardene [50] er det nødvendig å ha god kontroll på pulsformene som sendes ut. For å unngå beregningskompleksiteten som følger av et senderfilter av høy orden ble det besluttet å bruke en oppslagstabell med ferdig lagrede punktprøver, som skaleres for å gi ønsket amplitude og modulasjonsindeks. Kildekoden til den kombinerte senderblokken er vedlagt og nærmere beskrevet i tillegg A.3.

Pulsformene som brukes ble stilt til disposisjon av Q-Free, i form av en oppslagstabell med 512 punktprøver per symbol. Pulsformen på signalet og



Figur 7.3: Basisbåndsignalet $x[n]$ når datasekvensen 01010 er FM0-kodet ASK med pulsforming og en konstant likespenning. Symboltiden $T_s = 2$ μs, og det brukes 16 punktprøver per symbol. Verdien på punktprøvene må være innenfor ± 32767.

punktprøvene som overføres til USRP-en er vist i figur 7.3.

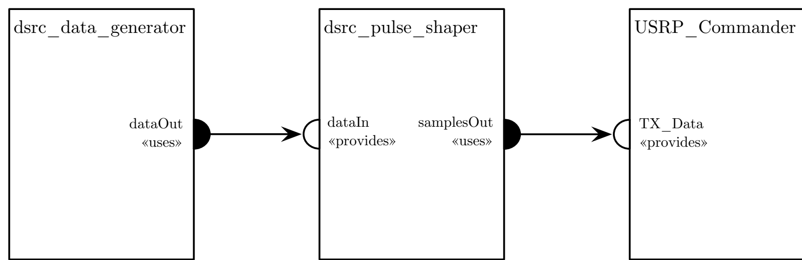
Det fremgår av likning (4.1) at den maksimale punktprøvingsraten over USB 2.0 er 16 megapunktprøver per sekund med reelle punktprøver. For senderen med symbol- og datarate på 500 kbit/s gir denne begrensningen maksimalt 32 punktprøver per symbol. Figur 7.3 viser pulsformen med 16 punktprøver per symbol, 8 ganger nyquistraten, hvilket ble vurdert som mer enn tilstrekkelig. Ytterligere interpolering foregår i USRP-en før digital-analog-omforming.

I denne implementasjonen tar pulsformerblokken den største mulige basisbåndamplituden U_{max} og den minste mulige basisbåndamplituden U_{min} i figur 7.3 som parametere. Sammen med amplitude på på bærebølgen i radioforsatsen, A_c, bestemmer disse modulasjonsindeksen, som vist i likning (3.7).

Senderflytskjemaet i Python er implementert slik at det tar diverse parametere på kommandolinjen, og det er mulig å spesifisere at datakilden skal være en datafil, en BST-sekvens, eller en ren sinusbølge. Det er også mulig å velge om dataene skal sendes ut på USRP-enheten, eller om de skal skrives til en fil. Flytskjemaet til senderen er vedlagt i tillegg A.1.

7.3.2 Sender implementert med OSSIE

Det ble laget en senderkomponent basert på blokken `dsrc_pulse_shaper_bs`, som ble utviklet for GNU Radio. Bølgeformen til senderen i OSSIE er satt sammen grafisk i Eclipse med OEF av senderkomponenten og en datagenerator som ble laget til dette prosjektet. Det ble benyttet en standardkomponent som håndterer data til og fra USRP-enheten, og gjør det mulig å spesifisere frekvensområder og interpolering på det utgående signalet. Komponentene som ble brukt, og måten de er koblet sammen på er vist i figur



Figur 7.4: Senderen realisert i OSSIE.

7.4. Koblingen til USRP og antennen er ellers som vist med GNU Radio i figur 7.2.

7.4 Mottaker

Det er to hovedmetoder for mottakerdesign når signalet er differensielt kodet: synkron, eller *koherent* deteksjon, som innebærer at mottakeren forsøker å spore den absolutte fasen til de mottatte datasymbolene; og differensiell, *ikke-koherent* deteksjon, hvor mottakeren bare ser på endring i fase fra et symbol til et annet. Ved ikke-koherent deteksjon vil støyvariansen være tilnærmet det dobbelte av ved koherent deteksjon, noe som vil gi rundt 3 dB dårligere støymargin [55].

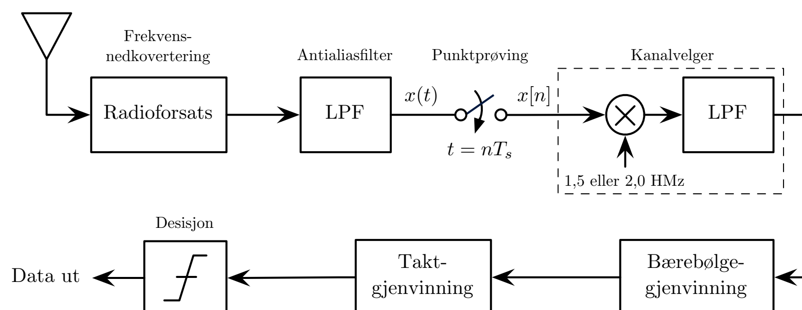
Begge metodene er behandlet under, hvor hovedfokuset er på den koherente mottakeren.

7.4.1 Koherent mottaker med GNU Radio

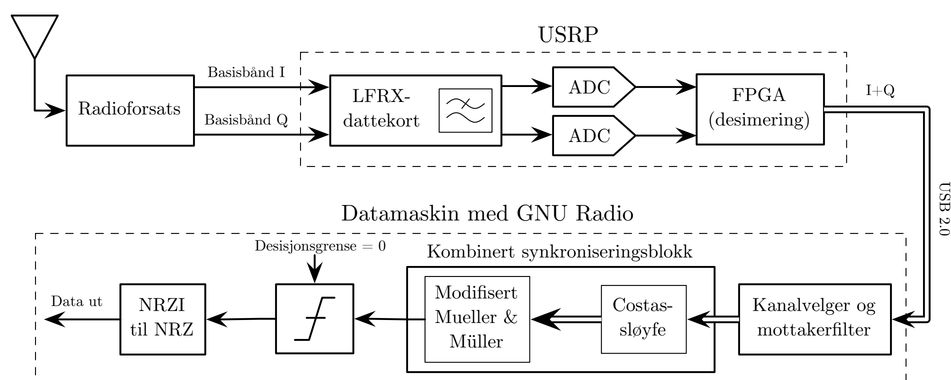
En generell koherent mottaker for veikantutstyret i DSRC vist i figur 7.5 på neste side. Den påfølgende beskrivelsen tar for seg hvordan denne generelle mottakeren er realisert med GNU Radio og USRP.

Det brukes en radioforsats fra Q-Free som gir ut to separate signaler i basisbånd via SMA-kontakter, fasekomponenten I, og kvadraturfasekomponenten Q. LFRX-kortet på USRP har to kanaler som analog-digital-omformes separat. Disse kombineres til én kompleks kanal i programvare. USRP-enheten og den radioforsatsen kobles derfor sammen med to kabler, som vist i figur 7.6 på neste side. Denne figuren viser signalveien i hele mottakeren.

Mottakeren er implementert som et flytskjema skrevet i Python med bruk av ferdige komponenter som er en del av GNU Radio-biblioteket. Det er blitt laget flere implementeringer med forskjellige sett av blokker, men denne rapporten behandler kun én variant av den koherente mottakeren.



Figur 7.5: Generell koherent mottaker. Her er det punktprøvede signalet $x[n] = x(nT_s)$, hvor T_s er tiden mellom hver punktprøve, reatert til punktprøvingsraten $f_s = \frac{1}{T_s}$.



Figur 7.6: Prinsipp for deteksjon og synkronisering slik det er gjort i denne realiseringen. Dobbel strek symboliserer et komplekst signal.

Mottakeren tar inn en flere parametere fra kommandolinjen, som inkluderer mulighet for å velge om kilden skal være USRP-enheten eller punktprøver lagret i en fil. Det er også mulig å velge en modus der utgangsdata fra mange blokker blir dumpet til fil for feilsøking. I tillegg velges frekvensen på det mottatte signalet, som typisk vil være 1,5 MHz eller 2 MHz i reelle anvendelser. Det er også mulig å motta rene basisbåndsignalet sentrert rundt 0 Hz, noe som i stor grad er blitt benyttet i forbindelse med testing. Python-koden til flytskjemaet er vedlagt i tillegg A.2.

USRP-kildeblokken er konfigurert til å kombinere de to I og Q-signalene på hver sin kanal («subdevice») på LFRX-kortet til ett komplekst signal. Desimeringen i FPGA spesifiseres i USRP-kildeblokken, og er valgt til 16, som gir $\frac{64\text{ MS/s}}{16} = 4$ MS/s. Ved en mottatt datarate på 250 kbit/s gir dette 16 punktprøver per symbol, men synkroniseringsblokkene kan fungere med så lite som som 2 punktprøver per symbol.

Kanalvelgeren er et filter som kombinerer nedkonvertering av spesifisert frekvens til basisbånd, og lavpassfiltrering med et FIR-filter. Det er forsøkt med flere typer filtre, men det mest fleksible er å bruke en filterdesignpakke i GNU Radio som tar inn vindustype, knekkfrekvens og filterorden som parametere, og genererer de nødvendige koeffisientene. Filterparametrene kan enkelt modifiseres i flytskjemaet.

Etter kanalvelgeren brukes det en innebygget bokk, `mpsk_receiver`, som kombinerer en Costas-sløyfe for bærebølgegjenvinning og en modifisert Mueller og Müller-algoritme for taktgjenvinning. Disse blokkene eksisterer også separat, men sammen gir de noe bedre ytelse i form av signal-støy-forhold, kombinert med lavere kompleksitet. Ut fra denne blokken kommer det én punktprøve per symbol. Teorien for Costas-sløyfen og den modifiserte Mueller og Müller-algoritmen ble presentert i henholdsvis kapittel 3.6 og 3.7.

Det neste skrittet i mottakeren er en desisjon, der desisjonsgrensen er satt til 0 ettersom begge symbolene er like sannsynlige. Det siste trinnet er en differensiell dekoder som konverterer fra NRZI til NRZ, vanlige binære data.

Betydningen av doppler og frekvensfeil i senderen i bilutstyret er diskutert under, med tanke på å finne parametere til synkroniseringsblokken i mottakeren. Frekvensområdet Costas-løkken kan operere i bestemmes av f_{min} og f_{max}. Ettersom signalet alltid er basisbånd vil dette ligge rundt 0, slik at $f_{min} = -f_{max}$. Det er viktig at området er stort nok til å få med sannsynlig frekvensvariasjon, men ikke så stort at man risikerer at sløyfen låser seg på et eventuelt annet signal på en annen frekvens.

Dopplerskift er definert som

$$f_d = f_0 \frac{v}{c} \cos\theta, \tag{7.1}$$

hvor f_d er dopplerfrekvensskiftet, f_0 er kildens frekvens, c er lyshastigheten, v er relativ hastighet mellom kilde og observatør, og θ er vinkelen mellom kilde og observatør. Dersom man antar at en bil beveger seg 80 km/t = 22,2 m/s rett mot antennen i veikantenheten, og at det sendes på en delkanal («subcarrier») på 1,5 MHz under hovedkanalen på 5,800 GHz, vil dopplerskiftet være

$$f_d = 5{,}7985 \text{ GHz} \cdot \frac{22{,}2 \text{ m/s}}{3 \cdot 10^8 \text{ m/s}} \cdot \cos 0 = 429 \text{ Hz}. \quad (7.2)$$

På grunn av at veikantutstyret bruker bærebølgen utsendt fra veikantutstyret, og kun genererer underkanalfrekvensen selv [56], vil frekvensavviket bli lite. Som vist i ligning (6.1) vil en maksimal frekvensfeil som er innenfor spesifikasjonene i DSRC-standarden være på 2000 Hz. Den totale frekvensvariasjonen vil da i verste fall bli i området $\Delta f = 2500$ Hz.

Punktprøvingsfrekvensen til AD-omformeren i USRP er $f_{s_1} = 64$ MS/s, og en desimeringsrate på 16 vil punktprøvingsfrekvensen inn på synkroniseringsblokken f_{s_2} være

$$f_{s_2} = \frac{f_{s_1}}{D} = \frac{64 \,\text{MS/s}}{16} = 4 \,\text{MS/s}. \quad (7.3)$$

Dermed vil en frekvensvariasjon på ± 2500 Hz tilsvare en normalisert frekvensvariasjon på

$$\Delta f_{norm} = \frac{2\pi \Delta f}{f_{s_2}} = \frac{2\pi \cdot 2500 \,\text{Hz}}{4 \,\text{MS/s}} = 3{,}927 \cdot 10^{-3}, \quad (7.4)$$

som er brukt i implementajsonen av mottakeren.

7.4.2 Alternativ ikke-koherent mottaker

Det er mulig å implementere mottakeren som en differensiell, ikke-koherent mottaker med enklere synkronisering. Dette er interessant for å undersøke forskjeller i beregningskompleksitet, og analysere ytelsesforskjellene med tanke på støymargin. Teorien for ikke-koherent mottak ble introdusert i kapittel 3.2 på side 7.

I tillegg til feilkildene vurdert i kapittel 7.4.1, vil faseendring som følge av endret avstand mellom senderen og mottakeren få betydning når det ikke brukes noen form for fasesynkronisering. Denne endringen vil likevel være liten når bilen beveger seg ved lav hastighet. I en ikke-koherent mottaker er det unødvendig å bruke beregningskraft på å følge den absolutte fasen til det mottatte signalet. Ettersom dataene er differensielt kodet er det kun endringen i fase mellom to påfølgende symboler som har betydning, hvilket gjør deler av synkroniseringen i den koherente mottakeren overflødig.

Det ble gjort forsøk på å implementere en ikke-koherent mottaker med GNU Radio, uten bruk av Costas-sløyfe for frekvens- og fasesynkronisering. I simuleringene ble det brukt et kjent basisbåndsignal hentet fra en vektorsignalgenerator ved hjelp av USRP, og lagret som punktprøver i en datafil. Med oppsettet som ble brukt genererte mM&M-synkroniseringsblokken færre punktprøver på utgangen enn forventet. Årsaken til dette ble ikke funnet, noe som kan skyldes at det ble dårlig tid til å forstå implementeringen av mM&M-blokken i GNU Radio i detalj. Som følge av problemene ble det ikke gjort tester av den ikke-koherente mottakeren med USRP og fysiske signaler.

7.5 Test og verfisering

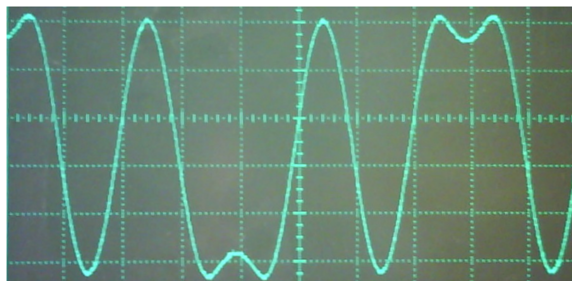
GNU Radio kommer med enkelte funksjoner for GNU Octave [57] og MATLAB som gjør det mulig å enkelt lese inn binærfiler med de forskjellige datatypene som brukes i GNU Radio, slik at de kan analyseres. Det er blitt brukt diverse utstyr i forbindelse med testing, blant annet en vektorsignalgenerator, oscilloskop, spektrumsanalysator og en 5,8 GHz mottaker som tar signalet ned til en mellomfrekvens slik at det kan visualiseres på et oscilloskop. Det ble også brukt en OBU-brikke med lysdioder som viser om den mottar en modulert bærebølge, og om den mottar en korrekt BST-ramme. Dersom brikken mottar en korrekt BST-ramme vil den svare, noe som kan brukes til å teste mottakeren i veikantutstyret.

7.5.1 Senderen implementert med GNU Radio

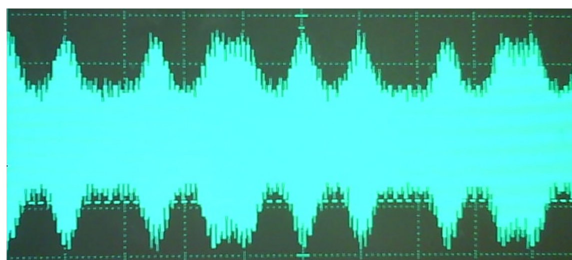
Funksjonaliteten som ligger i senderblokken `dsrc_pulse_shaper_bs` er blitt nøye testet ved hjelp av et omfattende testskript med forskjellige typer inndata. Blokken fungerer som den skal.

Under utprøvingen beskrevet under ble det brukt en punktprøvingsrate på 16 punktprøver per symbol, 8 MS/s. Selve flytskjemaet uten USRP ble testet ved å dumpe punktprøvene til fil, for så å plotte og analysere dem i MATLAB. Det neste skrittet i testingen var å måle på utgangen til USRP-enheten med et oscilloskop i basisbånd, før videre testing med radioforsatsen. Signalet i basisbånd, vist i figur 7.7 på neste side, er i tråd med det forventede, og bildet på oscilloskopet tilsvarer signalet vist i figur 7.3 på side 42.

Da utstyret var testet i basisbånd ble USRP-enheten koblet til radioforsatsen. Inngangstrinnet i radioforsatsen som var tilgjengelig var i uorden, noe som gjorde det vanskelig å kontrollere DC-nivået. Dette var en kjent feil med den aktuelle radioforsatsen, og medførte at det ikke var mulig å justere inn modulasjonsindeksen nøyaktig. Signalet på 5,8 GHz fra GNU Radio-realiseringen,



Figur 7.7: Basisbåndsignalet fra USRP målt med oscilloskop.



Figur 7.8: Utsendt passbåndsignal på 5,8 GHz nedkonvertert til en mellomfrekvens.

mottatt via radioforsatsen, ble nedkonvertert til en mellomfrekvens for å gjøre det mulig å vise signalet på et oscilloskop. Signalet er vist figur 7.8, og ser ut til å være som forventet, sett opp mot basisbåndsignalet i figur 7.7. Det er imidlertid tydelig at modulasjonsindeksen er noe lav, hvilket skyldes de nevnte problemene med radioforsatsen.

Det ble gjort forsøk med radioforsatsen i passbånd, med en testbrikke med lysdioder som viser om den mottar en modulert bærebølge og om den mottar en korrekt BST-ramme. En lysdiode på testbrikke indikerte at brikken våknet opp som følge av en modulert bærebølge. Imidlertid kjente ikke brikken igjen BST-sekvensen, som skulle ført til at en annen lysdioden blinket for å visualisere suksess. Dette gjorde også at brikken ikke svarte, noe som gjorde det umulig å teste mottakeren på denne måten.

Beregningskompleksiteten ble analysert ved å kjøre flytskjemaet med forskjellig antall punktprøver per sekund. Det ble analysert hvor store ressurser Python-prosessen opptok på testmaskinen. Denne prosessen inkluderer hele flytskemaet og alle blokkene, og gir derfor en god indikasjon på hvor mye realiseringen krever av prosesseringskraft. Dette tilsvarer metodene brukt av [44] i kapittel 5.3.1. Analysen ble utført ved hjelp av programmet *Aktivitetsmonitor*[1], som er en del av Mac OS X. Maskinvaren som ble brukt er

[1] Aktivitetsmonitor oppgir prosessorbelastningen i forhold til én prosessorkjerne, slik at prosessorbelastningen ifølge Aktivitetsmonitor kan bli opptil 200 % på testmaskinen med

Tabell 7.3: Beregningskompleksitet i senderen realisert med GNU Radio.

Punktprøver/symbol	Prosessorbelastning [%]	Minnebruk [MB]
4	3,6	15,4
8	5,7	15,4
16	10,5	15,4

beskrevet i tabell 7.1. Undersøkelsen viser at beregningskompleksiteten øker omtrent lineært med antall punktprøver per symbol når det brukes en fast datarate på 500 kbit/s. Resultatene er vist i tabell 7.3.

7.5.2 Senderen implementert med OSSIE

Det ble laget en komplett bølgeform med OSSIE-realiseringen der USRP-komponenten var erstattet med en komponent som skrev data til skjermen. Denne ble brukt i den initielle testingen, og senderkomponenten og bølgeformen forøvrig så ut til å fungere slik det var tenkt.

Ved fysiske tester med USRP var det ikke mulig å finne noe signal på utgangen av LFTX-datterkortet når bølgeformen ble kjørt. Årsaken til dette ble ikke funnet, men det har ikke vært mulig å bekrefte at LFTX-datterkortet fungerer med OSSIE. I et notat fra tidlig i 2007 er det nevnt at det var planer om å implementere støtte for LFTX-datterkortet [58], men det er vanskelig å finne kilder som kan bekrefte at det er blitt gjort. Undertegnede fikk tilbakemelding fra en OSSIE-utvikler om at datterkortet *burde* fungere [59], men har ikke vært i stand til å finne konkrete eksempler på at det stemmer. På bakgrunn av problemene ble det besluttet å ikke gjøre mer arbeid med OSSIE-realiseringen av senderen.

7.5.3 Koherent mottaker

Den koherente mottakeren er blitt testet i basisbånd ved hjelp av en vektorsignalgenerator koblet til USRP-enheten. Testingen viste at mottakeren fungerer som forventet i basisbånd, og at de mottatte dataene er identiske med de som ble sendt.

Det var planlagt å teste mottakeren ved å la senderen sende en BST-sekvens til en spesiell testbrikke, som så skulle svare med et kjent signal. På grunn av problemene med senderen som er beskrevet i kapittel 7.5.1 lot dette seg ikke gjennomføre, og den komplette mottakeren med radiofortsatsen er derfor ikke blitt testet i passbånd ved 5,8 GHz.

to kjerner. For å unngå uklarhet er derfor prosenttallene i tabell 7.3 og 7.4 dividert på 2, slik at 100 % representerer maksimal belastning på hele prosessoren.

Tabell 7.4: Beregningskompleksitet i den koherente mottakeren realisert med GNU Radio.

Punktprøver/symbol	Prosessorbelastning [%]	Minnebruk [MB]
4	11,2	15,6
8	22,2	15,6
16	51,4	15,6

Beregningskompleksiteten ble analysert ved å kjøre flytskjemaet med forskjellig antall punktprøver per sekund, med tilsvarende fremgangsmåte som for senderen i kapittel 7.5.1. Resultatene av undersøkelsen er vist i tabell 7.4. Det fremgår at beregningskompleksiteten øker tilnærmet lineært med antall punktprøver per symbol når det brukes en fast datarate på 250 kbit/s. Beregningskompleksiteten er som forventet betydelig høyere for mottakeren enn for senderen.

7.6 Beregningskompleksitet

Analysene av beregningskompleksitet viser at det verste tilfellet testet, mottakeren med 16 punktprøver per symbol, krever i overkant av 50 % av prosesseringskraften tilgjengelig i testmaskinen. En prototype på en integrert SDR basert på USRP og GNU Radio [11] har spesifikasjoner som tilsvarer datamaskinen som ble brukt til testing i dette prosjektet, og har tilstrekkelig med datakraft. Dimensjonene på denne prototypen er $29 \times 27 \times 21$ cm, med en vekt på rundt 3 kilogram. Dette viser at det er mulig å lage en integrert enhet med USRP og en enkel datamaskin som kan fungere som en komplett DSRC-veikantenhet sammen med en ekstern radioforsats.

Kapittel 8

Konklusjon

Denne masteroppgaven har undersøkt hvordan hyllevarekomponenter og frie rammeverk for programvaredefinert radio kan anvendes i forbindelse med dedikert kortholdslink (DSRC). Det innledende litteraturstudiet har vist at GNU Radio er det frie rammeverket som er best egnet for en programvaredefinert realisering av fysisk lag i DSRC. Det er mulig å bruke GNU Radio sammen med høyere lags protokoller implementert på siden, men det vil ikke være trivielt å implementere disse i GNU Radio med den nåværende arkitekturen. OSSIE er på grunn av høy latens mellom komponentene uegnet til duplekssystemer med krav til kort forsinkelse, deriblant DSRC.

Sender- og mottakersiden til en veikantenhet for medium datarate er blitt delvis implementert hver for seg på en datamaskin med GNU Radio og USRP. Senderen ser ut til å fungere som den skal når det måles på utgangssignalet, men under testing med en bilbrikke var det ikke mulig å oppnå kommunikasjon. En feil med den aktuelle radioforsatsen, som gjorde det vanskelig å justere modulasjonsindeksen, kan være en del av forklaringen på problemet. Mottakeren ble ble kun testet i basisbånd, men ser ut til å fungere.

En OSSIE-realisering av senderen fungerer ikke, noe som kan skyldes problemer med dårlig støtte datterkortet LFTX i OSSIE. Det var derfor ikke mulig å sammenligne ytelsen til de to realiseringene av senderen opp mot hverandre gjennom praktiske forsøk.

GNU Radio-realiseringene har en moderat beregningskompleksitet, hvilket bør gjøre det mulig å integrere USRP-hovedkortet og datterkort sammen med en enkel datamaskin i en liten enhet. Sammen med en radioforsats og nødvendig programvare vil dette kunne utgjøre en komplett og selvstendig veikantenhet.

Det gjenstår arbeid med å verifisere funksjonaliteten til både senderen og mottakeren, og med å optimalisere parametrene i synkroniseringsalgoritmene og filtrene i mottakeren. Det bør arbeides videre med å implementere en ikke-koherent mottaker, som antakelig vil være langt mindre beregningsintensiv som følge av enklere synkronisering. For å lage en komplett fungerende veikantenhet, kan det være interessant å se på mulighetene til å knytte fysisk lag utviklet i dette prosjektet til eksisterende implementeringer av høyere lags protokoller. En del av denne enheten kan bestå av datterkortet XCVR2450, som dekker hele ISM-båndet ved 5,8 GHz, og kan erstatte mye av radioforsatsen fra Q-Free som er brukt i den nåværende realiseringen.

Bibliografi

[1] E. Thorsrud, "Programvaredefinert radio: realisering av et STANAG 4285-sendermodem." Fordypningsprosjekt, desember 2008.

[2] C. Aasen, "Forsker på trådløst babeltårn," *Computerworld*, September 2006.

[3] S. Ellingson and S. S. Hasan, "The rise of all-band all-mode radio," Technical Memo No. 17, Virginia Polytechnic Institute & State University, Blacksburg, Virginia, January 2007.

[4] J. Mitola, "Software radios: Survey, critical evaluation and future directions," *Telesystems Conference, 1992. NTC-92., National*, pp. 13/15–13/23, May 1992.

[5] S. Haykin, "Cognitive radio: brain-empowered wireless communications," *IEEE Journal on Selected Areas in Communications*, vol. 23, pp. 201–220, February 2005.

[6] W. H. W. Tuttlebee, *Software Defined Radio: Enabling Technologies.* New York: J. Wiley & Sons, 2002.

[7] W. H. W. Tuttlebee, *Software Defined Radio: Origins, Drivers, and International Perspectives.* West Sussex, England: John Wiley, 2002.

[8] H. Arslan, ed., *Cognitive Radio, Software Defined Radio, and Adaptive Wireless Systems.* Tampa: Springer, 2007.

[9] SDR Forum, "Overview and Definition of Software Download for RF Reconfiguration." Document SDRF-02-P-0002-V1.0.0, August 2002.

[10] R. R. Tummala, "Moore's law meets its match (system-on-package)," *Spectrum, IEEE*, vol. 43, no. 6, pp. 44–49, 2006.

[11] M. Dickens, B. Dunn, and L. J. Nicholas, "Design and Implementation of a Portable Software Radio," *Communications Magazine, IEEE*, vol. 46, no. 8, pp. 58–66, 2008.

[12] S. S. Haykin, *Communication systems.* New York: Wiley, fourth ed., 2001.

[13] J. P. Costas, "Synchronous Communications," *Proceedings of the IEEE*, vol. 90, pp. 1461–1466, August 2002.

[14] J. Feigin, "Practical costas loop desig," *Electronic Design Group*, January 2002.

[15] D. P. Taylor, "Introduction to Synchronous Communications"," *Proceedings of the IEEE*, vol. 90, pp. 1459–1460, August 2002.

[16] G. R. Danesfahani and T. G. Jeans, "Optimisation of modified Mueller and Müller algorithm," *Electronics Letters*, vol. 31, pp. 1032–1033, June 1995.

[17] F. A. Hamza, "The USRP under 1.5X Magnifying Lens." `http://www.gnuradio.org/trac/attachment/wiki/UsrpFAQ/USRP_Documentation.pdf?format=raw`, June 2008. Rev 1.0.

[18] T. Danielsen, "Software-Defined GNSS Receiver based on Free Software Components," Master's thesis, Norwegian University of Science and Technology, Trondheim, July 2007.

[19] E. Hogenauer, "An economical class of digital filters for decimation and interpolation," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 29, no. 2, pp. 155–162, 1981.

[20] Q. Norton, "GNU Radio Opens an Unseen World." `http://www.wired.com/science/discoveries/news/2006/06/70933`, May 6 2006.

[21] P. Balister and H. Reed, "USRP Hardware and Software Description," Technical Memo No. 9, Virginia Polytechnic Institute & State University, Blacksburg, Virginia, June 2006.

[22] "GNU Radio." `http://gnuradio.org`.

[23] Ettus Research LLC, "USRP2 Datasheet." Sales Brochure.

[24] J. G. Proakis and D. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications.* Upper Saddle River, N.J.: Pearson Prentice Hall, fourth ed., 2007.

[25] T. Schmid, O. Sekkat, and M. B. Srivastava, "An experimental study of network performance impact of increased latency in software defined

radios," in *WinTECH '07: Proceedings of the the second ACM international workshop on Wireless network testbeds, experimental evaluation and characterization*, (New York, NY, USA), pp. 59–66, ACM, 2007.

[26] Ettus Research LLC, "TX and RX Daughterboards For the USRP Software Radio System." Sales Brochure.

[27] Ettus Research LLC, "Transceiver Daughterboards For the USRP Software Radio System." Sales Brochure.

[28] Free Software Foundation, "GNU General Public License." `http://www.gnu.org/copyleft/gpl.html`.

[29] E. Blossom, "GNU Radio: Tools for Exploring the Radio Frequency Spectrum," *Linux Journal*, vol. 122, June 2004.

[30] B. Stroustrup, *The C++ programming language.* Reading, Mass.: Addison-Wesley, special ed., 2000.

[31] R. Dhar, G. George, A. Malani, and P. Steenkiste, "Supporting Integrated MAC and PHY Software Development for the USRP SDR," *SDR '06 1st IEEE Workshop on Networking Technologies for Software Defined Radio Networks*, pp. 68–77, 2006.

[32] GNU Radio Wiki, "How to Write GNU Radio Python Applications." `http://gnuradio.org/trac/wiki/Tutorials/WritePythonApplications`.

[33] H. von Malm, "Implementing physical and data link control layer on the GNU software-defined radio platform." Universität Paderborn, Studienarbeit, December 2005.

[34] E. Blossom, "Re: [Discuss-gnuradio] GNU Radio Release 3.2 available for download or binary installation." `http://lists.gnu.org/archive/html/discuss-gnuradio/2009-05/msg00457.html`, May 2009.

[35] E. Blossom, "[Discuss-gnuradio] extract timestamp data / The Plan." `http://www.mail-archive.com/discuss-gnuradio@gnu.org/msg18000.html`, March 2009.

[36] A. Feickert, "The Joint Tactical Radio System (JTRS) and the Army's Future Combat System (FCS): Issues for Congress." CRS Report for Congress, November 2005.

[37] Joint Program Executive Office, Joint Tactical Radio System, *Software Communications Architecture specification. Version 2.2.2*, 15. Mai 2006.

[38] J. Bard and V. J. Kovarik, *Software Defined Radio: The Software Communications Architecture.* Chichester, West Sussex, England: John Wiley, 2007.

[39] "The OMG's CORBA Website." `http://www.corba.org/`.

[40] T. Sundquist, "Waveform development using software defined radio," Master's thesis, Linköpings Universitet, Norrköping, Sverige, April 2006.

[41] M. Carrick, D. Cormier, C. Covington, C. B. Dietrich, J. Gaeddert, B. Hilburn, C. I. Phelps, S. Sayed, J. Snyder, and H. Volos, *OSSIE 0.7.2 Installation and User Guid*, November 2009.

[42] OSSIE Team, "Open Source SCA Implementation – Embedded." `http://ossie.mprg.org`.

[43] P. J. Balister, "A Software Defined Radio Implemented using the OSSIE Core Framework Deployed on a TI OMAP Processor," Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, December 2007.

[44] G. Abgrall, F. L. Roy, J.-P. Delahaye, J.-P. Diguet, and G. Gogniat, "A comparative study of two software defined radio environments," in *Proceedings of the SDR '08 Technical Conference and Product Exposition*, 2008.

[45] J. O. Neset, "Software Defined Radio – Analysis of protability issues using Open Source SCA Imlementation – Embeded," Master's thesis, Norwegian University of Science and Technology, June 2008.

[46] Álvaro Palomo Navarro, R. Villing, and R. Farrell, "Software defined radio architectures evaluation," in *Proceedings of the SDR '08 Technical Conference and Product Exposition*, 2008.

[47] H.-J. Fischer, "Dedicated Short Range Communication (DSRC) – A Tutorial," July 2003.

[48] M. Y. Nygård, "Betal bilvask og bensin med bombrikken," *Adresseavisen*, February 2, 2009.

[49] J. A. Audestad, *Technologies and systems for access and transport networks.* Boston: Artech House, 2008.

[50] European Committee for Standardization, "EN 12253: Road transport and traffic telematics — Physical layer using microwave at 5.8 GHz," July 2004.

[51] European Committee for Standardization, "EN 12795: Road transport and traffic telematics — Deticated Short Range Communication (DSRC) — DSRC data link layer: medium access and logical link control," March 2003.

[52] European Committee for Standardization, "EN 12834: Road transport and traffic telematics — Deticated Short Range Communication (DSRC) — DSRC application layer," November 2003.

[53] European Committee for Standardization, "EN 13372: Road transport and traffic telematics — Deticated Short Range Communication — Profiles for RTTT applications," July 2003.

[54] International Organization for Standardization, "ISO/TS 14907-1: Road transport and traffic telematics — Electronic fee collection — Test procedures for user and fixed equipment — Part 1: Description of test procedures," February 2005.

[55] J. R. Barry, E. A. Lee, and D. G. Messerschmitt, *Digital communication*. Boston: Kluwer Academic Publishers, third ed., 2004.

[56] R. Prata, N. Fernandes, A. Serrador, and F. Fortes, "On Board Equipment for DSRC systems," in *6th conference on telecommunications*, (Peniche, Portugal), May 2007.

[57] "GNU Octave." `http://www.gnu.org/software/octave/`.

[58] Wireless @ Virginia Tech, "News and Notes for our Industrial Partners." `http://wireless.vt.edu/affiliates/newsletters/feb_07.pdf`, February 2007.

[59] C. Phelps, "[Discuss-OSSIE] Re: USRP LFTX/LFRX daughter cards." `http://listserv.vt.edu/cgi-bin/wa?A2=ind0903&L=ossie-discuss&T=0&P=2981`, March 2009.

Tillegg A

Kildekode

Dette tillegget inneholder Python-koden til GNU Radio flytskjemaene. For å spare plass inneholder tillegget ikke kildekoden til blokkene som er utviklet i forbindelse med prosjektet, med unntak av senderkomponenten ettersom den håndterer mesteparten av signalbehandlingen i senderen alene, og er et godt eksempel på hvordan komponenter utvikles i GNU Radio.

A.1 Senderen realisert med GNU Radio

Senderflytskjemaet som er listet opp under er beskrevet i kapittel 7.4.1. Python-filen som inneholder flytskjemaet er vist i listing A.1.

Listing A.1: `dsrc_rsu_transmitter.py`

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  ##################################################
4  # GNU Radio Python Flow Graph
5  # Title: DSRC RSU Transmitter
6  # Author: Einar Thorsrud
7  # Description: Transmitter for the DSRC RSU using the USRP.
8  ##################################################
9
10 from gnuradio import gr
11 from gnuradio import eng_notation
12 from gnuradio.eng_option import eng_option
13 from grc_gnuradio import usrp as grc_usrp
14 from gnuradio import dsrc
15 from optparse import OptionParser
16 import time
17
18 class top_block(gr.top_block):
19
20   def __init__(self):
21     gr.top_block.__init__(self, "DSRC RSU Transmitter")
```

```
22
23        ##########################################################
24        # Options
25        ##########################################################
26        parser = OptionParser(option_class=eng_option)
27
28        parser.add_option("-i", "--input-file", type="string", default="
              binary_nrz_data.dat",
29          help="specify input-data-filename [default=%default]")
30
31        parser.add_option("-o", "--output-file", type="string", default="
              output_samples.dat",
32          help="specify output-data-filename [default=%default]")
33
34        parser.add_option("-s", "--sink", type="string", default="usrp",
35          help="specify the sink (\"usrp\" or \"file\") [default=%default]
                ")
36
37        parser.add_option('-r', '--repeat', action="store_true", default=
              False,
38          help="repeat input file or BST sequence")
39
40        parser.add_option('-c', '--carrier', action="store_true", default=
              False,
41          help="pure sine wave")
42
43        parser.add_option('-b', '--bst', action="store_true", default=
              False,
44          help="send a BST sequence every 10 millisecond")
45
46        (options, args) = parser.parse_args ()
47        if len(args) != 0:
48          parser.print_help()
49          raise SystemExit, 1
50
51        self.input_file = options.input_file
52        self.output_file = options.output_file
53        self.sink = options.sink
54        self.repeat = options.repeat
55        self.carrier = options.carrier
56        self.bst = options.bst
57
58        ##########################################################
59        # Variables
60        ##########################################################
61        self.frequency = 0
62        self.gain = 1.0 # USRP transmitter gain
63        self.bit_per_second = 500e3 # For MDR this is 500 kbit/s
64        self.usrp_samples_per_second = 128e6 # The USRP DAC is 128 MS/s
65        self.frequency = 0
66
67        # 25000 max and -300 was OK with a deffect RF-frontend
68        self.v_min = 25000  # within the range -32768/+32767
69        self.v_max = 1000 # within the range -32768/+32767
70        self.phase = 1 # +1 or -1
71        self.pulse_shaper_interpolation = 16
72
73        # Variable calculated based on the constants above
74        self.interpolation_rate = 2 * self.usrp_samples_per_second / (self
              .bit_per_second * self.pulse_shaper_interpolation)
75
76        ##########################################################
```

```
77        # Generic blocks (Non USRP) and connections
78        ##################################################
79        self.pulse_shaper = dsrc.pulse_shaper_bs (self.v_min, self.v_max,
              self.phase, self.pulse_shaper_interpolation)
80
81        ##################################################
82        # Generic blocks (Non USRP) and connections
83        ##################################################
84        if self.carrier == True:
85          self.source = gr.vector_source_b((0, 0, 0, 0), True) # Sine at
                500 kHz
86
87        elif self.bst == True:
88          self.preamble = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
89
90          self.bst_sequence = (
91            0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, \
92            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, \
93            0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, \
94            0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, \
95            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 )
96
97          # Sending the preamble 312 first to wake up the OBU, then a BST
98          self.source = gr.vector_source_b(self.preamble * 312 + self.
                bst_sequence + self.preamble, self.repeat)
99
100       else:
101         self.source = gr.file_source(gr.sizeof_char*1, self.input_file,
                self.repeat)
102
103       ##################################################
104       # USRP sink blocks
105       ##################################################
106       if self.sink.lower() == 'usrp':
107         self.sink = grc_usrp.simple_sink_s(which=0, side='A')
108         self.sink.set_interp_rate(self.interpolation_rate)
109         self.sink.set_frequency(self.frequency, verbose=False)
110         self.sink.set_gain(self.gain)
111
112       ##################################################
113       # File sink block
114       ##################################################
115       else:
116         self.sink = gr.file_sink(gr.sizeof_short, self.output_file)
117
118       ##################################################
119       # Connections
120       ##################################################
121       self.connect((self.source, 0), (self.pulse_shaper, 0))
122       self.connect((self.pulse_shaper, 0), (self.sink, 0))
123
124
125 if __name__ == '__main__':
126
127   tb = top_block()
128
129   if tb.bst:
130     if tb.repeat:
131       tb.start()
132       raw_input('Press Enter to quit: ')
133       tb.stop()
134     else:
```

```
135         tb.run()
136
137
138     elif tb.repeat:
139       tb.start()
140       raw_input('Press Enter to quit: ')
141       tb.stop()
142
143     else:
144       tb.run()
```

A.2 Mottakeren realisert med GNU Radio

Mottakerflytskjemaet som er listet opp under er beskrevet i kapittel 7.4.1. Python-filen som inneholder flytskjemaet er vist i listing A.2.

Listing A.2: `dsrc_rsu_receiver.py`

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  ##################################################
4  # GNU Radio Python Flow Graph
5  # Title: DSRC RSU Receiver
6  # Author: Einar Thorsrud
7  # Description: Receiver for the DSRC RSU using the USRP.
8  ##################################################
9
10 from gnuradio import gr
11 from gnuradio import eng_notation
12 from gnuradio.eng_option import eng_option
13 from gnuradio import usrp
14 from usrpm import usrp_dbid
15 from grc_gnuradio import usrp as grc_usrp
16 from optparse import OptionParser
17 import time
18 from gnuradio import dsrc
19
20 class top_block(gr.top_block):
21
22   def __init__(self):
23     gr.top_block.__init__(self, "DSRC RSU Receiver path")
24
25     ##################################################
26     # Options
27     ##################################################
28     parser = OptionParser(option_class=eng_option)
29
30     parser.add_option("-o", "--output-file", type="string", default="
           binary_nrz_data.dat",
31       help="specify output-data-filename [default=%default]")
32
33     parser.add_option("-d", "--duration", type="eng_float", default=0,
34       help="specify the duration in milliseconds [default=%default]")
35
36     parser.add_option("-f", "--subcarrier-freq", type="eng_float",
           default=0,
37       help="specify the subcarrier frequency in Hertz [default=%
             default]")
```

```
38
39      parser.add_option("-s", "--source", type="string", default="usrp",
40        help="specify the source (\"usrp\" or \"file\") [default=%
            default]")
41
42      parser.add_option('-v', '--verbose', action="store_true", default=
            False,
43        help="verbose output")
44
45      parser.add_option('-r', '--repeat', action="store_true", default=
            False,
46        help="repeat input file sequence (if used)")
47
48      (options, args) = parser.parse_args ()
49      if len(args) != 0:
50        parser.print_help()
51        raise SystemExit, 1
52
53      self.output_file = options.output_file
54      self.duration_in_ms = options.duration
55      self.frequency = options.subcarrier_freq
56      self.source = options.source
57      self.verbose = options.verbose
58      self.repeat = options.repeat
59
60      ###################################################
61      # Variables and constants
62      ###################################################
63      self.samp_per_symb = 16 # Changes the amount of data over USB
64      self.gain = 1.0 # Gain in the USRP
65      self.symbol_per_second = 250e3 # Symbol rate equals the bit rate
66      self.usrp_samples_per_second = 64e6
67      self.input_rate = int(self.samp_per_symb * self.symbol_per_second)
68      self.decimation_rate = int(self.usrp_samples_per_second / self.
            input_rate)
69
70      # Variable for the MPSK-receiver synchronization block:
71      M = 2 # The modulation order of the MPSK modulation
72      costas_theta = 0
73      costas_alpha = 0.01
74      costas_beta = 0.000025
75      costas_fmin = -0.00393 # Min normalized frequency
76      costas_fmax = 0.00393 # Max normalized frequency
77      m_and_m_mu = 0.5 # Start value [0,1]
78      m_and_m_gain_mu = 0.05 # Gain parameter of the M&M signal
79      m_and_m_omega = self.samp_per_symb
80      m_and_m_gain_omega = (m_and_m_omega * m_and_m_omega) / 4 # Loop
            gain (Beta)
81      m_and_m_omega_rel = 0.005
82
83      self.lowpass_coeff = gr.firdes.low_pass (
84        1.0,           # gain
85        self.input_rate,  # sampling rate
86        100e3,          # low pass cutoff freq
87        100e3,          # width of trans. band
88        gr.firdes.WIN_HAMMING) # Window
89
90
91      ###################################################
92      # Generic blocks (Non USRP)
93      ###################################################
94
```

```
95      # synchronization block (Costas and M&M)
96      self.mpsk_receiver = gr.mpsk_receiver_cc(
97        M,
98        costas_theta,
99        costas_alpha,
100       costas_beta,
101       costas_fmin,
102       costas_fmax,
103       m_and_m_mu,
104       m_and_m_gain_mu,
105       m_and_m_omega,
106       m_and_m_gain_omega,
107       m_and_m_omega_rel)
108
109     # Extract data by hard decision:
110     self.complex_to_real = gr.complex_to_real(1) # Remove imaginary
          part
111     self.binary_slicer = gr.binary_slicer_fb() # Decision device
112
113     # NRZI to NRZ (differential decoding)
114     self.nrzi_to_nrz = dsrc.nrzi_to_nrz_bb (1) # Initial state 1 or 0
115
116     # Frequency translating low pass FIR filter for channel selection
117     # self.channel_filter = gr.fir_filter_ccf(1, self.lowpass_coeff)
118     self.channel_filter = \
119       gr.freq_xlating_fir_filter_ccf (
120         1, # 1 = no decimation
121         self.lowpass_coeff, # Channel filter coef
122         self.frequency,    # Station frequency
123         self.input_rate)     # Input sampling rate
124
125
126     # File sink for the decoded data
127     self.file_sink_decoded_data = gr.file_sink(gr.sizeof_char, self.
          output_file)
128
129
130     ##################################################
131     # USRP blocks and connections
132     ##################################################
133     if self.source.lower() == 'usrp':
134       self.usrp_source = usrp.source_c(0, self.decimation_rate)
135       self.subdev_spec = (0,2) # Side A, sub device A and B combined
136       self.usrp_source.set_mux(usrp.determine_rx_mux_value(self.
            usrp_source, self.subdev_spec))
137       self.subdev = usrp.selected_subdev(self.usrp_source, self.
            subdev_spec)
138       self.usrp_source.set_rx_freq(0, 0.0) # Set frequency 0.0 Hz on
            board number 0
139       self.subdev.set_gain(self.gain)
140
141       self.connect((self.usrp_source, 0), (self.channel_filter, 0))
142
143
144     ##################################################
145     # File source blocks and connections
146     ##################################################
147     else: #if self.source.lower() != 'usrp':
148       # Block to read file with complex samples when not using the
            USRP
149       # False/True decided whether to repeat indefinitely or not
150       self.file_source = gr.file_source(gr.sizeof_gr_complex*1, "
```

```
            raw_input_samples.dat", self.repeat)
151
152         self.connect((self.file_source, 0), (self.channel_filter, 0))
153
154
155       #####################################################
156       # Verbose blocks and connections (log to files)
157       #####################################################
158       if self.verbose:
159         self.file_sink_samples = gr.file_sink(gr.sizeof_gr_complex, "
            complex_raw_samples.dat")
160         self.file_sink_filtered_samples = gr.file_sink(gr.
            sizeof_gr_complex, "complex_filtered_samples.dat")
161         self.file_sink_symbols = gr.file_sink(gr.sizeof_gr_complex, "
            complex_symbols.dat")
162         self.file_sink_data = gr.file_sink(gr.sizeof_char, "
            binary_nrzi_data.dat")
163
164
165         if self.source.lower() == 'usrp':
166           self.connect((self.usrp_source, 0), (self.file_sink_samples,
              0))
167         else:
168           self.connect((self.file_source, 0), (self.file_sink_samples,
              0))
169
170         self.connect((self.channel_filter, 0), (self.
            file_sink_filtered_samples, 0))
171
172         # For writing synchronized symbols to file
173         self.connect((self.mpsk_receiver, 0), (self.file_sink_symbols,
            0))
174
175         # For writing binary data to file
176         self.connect((self.binary_slicer, 0), (self.file_sink_data, 0))
177
178
179
180       #####################################################
181       # Other connections
182       #####################################################
183       # Connect file_source to synchronization block:
184       self.connect((self.channel_filter, 0), (self.mpsk_receiver, 0))
185
186       # Connect symbols to binary data
187       self.connect((self.mpsk_receiver, 0), (self.complex_to_real, 0))
188       self.connect((self.complex_to_real, 0), (self.binary_slicer, 0))
189
190       # Convert NRZI to NRZ
191       self.connect((self.binary_slicer, 0), (self.nrzi_to_nrz, 0))
192
193       # Write binary data to file
194       self.connect((self.nrzi_to_nrz, 0), (self.file_sink_decoded_data,
          0))
195
196
197 if __name__ == '__main__':
198
199   # Starting flowgraph
200   tb = top_block()
201
202   if tb.duration_in_ms > 0: # Duration has been specified
```

```
203        print "Receiving for " + str(tb.duration_in_ms) + " milliseconds
              ..."
204        tb.start()
205        # Wait specified ammount of time befor terminating the flowgraph
206        time.sleep(tb.duration_in_ms/1000.0)
207        tb.stop()
208
209      elif tb.source.lower() != 'usrp': # File source
210        tb.run()
211
212      else: # USRP source, receving untill terminated
213        tb.start()
214        raw_input('Press Enter to quit: ')
215        tb.stop()
216
217      print "\nFinished! Output data is stored in: " + tb.output_file
```

A.3 Pulsformer-blokken i GNU Radio

GNU Radio-blokkene som er utviklet i dette prosjektet er del av av en modul kalt `dsrc`, som samler alt i en pakke. Det innebærer at blokkene kompileres og installeres sammen. Det er laget en serie med testskript som brukes til å verifisere at pakken og alle blokkene fungerer som de skal etter kompilering. DSRC-modulen inneholder enkelte blokker ut over pulsformeren, blant annet blokker for konvertering mellom NRZ og NRZI, men disse er ikke listet opp i dette tillegget. Blokkene er basert på en eksempelblokk lisensiert under «GNU General Public License» (GPL). Ettersom GPL er smittsom [28] er blokkene utviklet i denne masteroppgaven underlagt GPL.

Senderfunkjsonaliteten med FM0-koding, modulasjon og pulsforming med variabel modulasjonsindeks, er innbakt i `dsrc_pulse_shaper_bs`. Denne er basert på, og arver sine egenskaper fra, `gr_sync_interpolator`, som er en av de grunnleggende klassene for blokker i GNU Radio. Som de to siste bokstavene i navnet indikerer tar blokken inn data av typen `byte`, som er en 8 bit `char`. Her er informasjonen ett bit som er lagret i LSB, slik at verdien er enten 1 eller 0. Blokken sender ut data av typen `short`, som er heltallsverdier med fortegn.

I tillegg til inn og utgangsportene har blokken fire parametere. Disse er beskrevet under.

`v_max` brukes til å definere den høyeste signalverdien, U_{max}.

`v_min` brukes til å definere den laveste signalverdien, U_{min}. Både U_{max} og U_{min} må være innenfor ± 32767.

`preload` brukes til å definere den siste signalverdien før de første dataene som behandles. Den forrige verdien må være kjent på grunn av den differensielle FM0-kodingen, og er nyttig i forbindelse med testing.

`interpolation` bestemmer antallet punktprøver per symbol. Dette tallet må være delelig på to, og være mellom 4 og 512.

Blokken har lagret en halv periode av hvert av de to symbolene. Ved å snu på halvperiodene er det nok til å lage en hel periode, og både positivt og negativt fortegn, som illustrert i figur 7.3 på side 42. Det er lagret 256 punktprøver i hver halvperiode, som gir totalt 512 punktprøver per symbol. Dette er det som bestemmer den maksimale interpolasjonsfaktoren. Det plukkes ut et gitt antall symboler, avhengig av interpolasjonsfaktoren, og de skaleres og det adderes en konstant likespenningskomponent, i tråd med `v_max`- og `v_min`-parametrene.

C++-filen som inneholder det meste av funksjonaliteten er vist i listing A.3.

Listing A.3: `dsrc_pulse_shaper_bs.cc`

```
1  /* -*- c++ -*- */
2  /*
3   * Copyright 2009 Einar Thorsrud.
4   *
5   * This file is part of the GNU Radio DSRC module.
6   *
7   * The GNU Radio DSRC module is free software; you can redistribute
8   * it and/or modify it under the terms of the GNU General Public
9   * License as published by the Free Software Foundation; either
10  * version 2, or (at your option) any later version.
11  *
12  * The GNU Radio DSRC module is distributed in the hope that it will
13  * be useful, but WITHOUT ANY WARRANTY; without even the implied
14  * warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
15  * See the GNU General Public License for more details.
16  *
17  * You should have received a copy of the GNU General Public License
18  * along with the GNU Radio DSRC module; see the file COPYING.  If
19  * not, write to the Free Software Foundation, Inc., 51 Franklin
20  * Street, Boston, MA 02110-1301, USA.
21  */
22
23 #ifdef HAVE_CONFIG_H
24 #include "config.h"
25 #endif
26
27 #include <dsrc_pulse_shaper_bs.h>
28 #include <gr_io_signature.h>
29 #include <stdexcept>
30 #include <assert.h>
31
32 /*
33  * Create a new instance of dsrc_fm0_modulator_bc and return
34  * a boost shared_ptr.  This is effectively the public constructor.
35  */
36 dsrc_pulse_shaper_bs_sptr
37 dsrc_make_pulse_shaper_bs (short v_max, short v_min, int preload, int
     interpolation)
38 {
39   return dsrc_pulse_shaper_bs_sptr (new dsrc_pulse_shaper_bs (v_max,
       v_min, preload, interpolation));
40 }
41
```

```
42
43  static const int MIN_IN = 1;  // mininum number of input streams
44  static const int MAX_IN = 1;  // maximum number of input streams
45  static const int MIN_OUT = 1; // minimum number of output streams
46  static const int MAX_OUT = 1; // maximum number of output streams
47
48  dsrc_pulse_shaper_bs::dsrc_pulse_shaper_bs (short v_max, short v_min,
        int preload, int interpolation)
49    : gr_sync_interpolator ("fm0_modulator_bc",
50                           gr_make_io_signature (MIN_IN, MAX_IN, sizeof
                                 (unsigned char)),
51                           gr_make_io_signature (MIN_OUT, MAX_OUT,
                                 sizeof (short)),
52                           interpolation),
53                max(v_max),
54                min(v_min),
55                sign(preload),
56                interpol_fac(interpolation)
57
58  {
59
60  }
61
62
63  int
64  dsrc_pulse_shaper_bs::work (int                        noutput_items,
65                           gr_vector_const_void_star &input_items,
66                           gr_vector_void_star       &output_items)
67  {
68
69    const int HALF_SIN_LENGTH = 256;
70
71    short half_sin[HALF_SIN_LENGTH] = {
72         0,   25,   50,   75,  100,  125,  150,  175,
73       200,  225,  250,  275,  300,  325,  349,  374,
74       399,  423,  448,  472,  497,  521,  545,  569,
75       593,  617,  641,  665,  688,  712,  735,  759,
76       782,  805,  828,  851,  874,  896,  919,  941,
77       963,  985, 1007, 1029, 1050, 1072, 1093, 1114,
78      1135, 1155, 1176, 1196, 1216, 1236, 1256, 1276,
79      1295, 1315, 1334, 1352, 1371, 1389, 1408, 1426,
80      1443, 1461, 1478, 1495, 1512, 1529, 1545, 1561,
81      1577, 1593, 1608, 1624, 1639, 1653, 1668, 1682,
82      1696, 1710, 1723, 1736, 1749, 1762, 1774, 1786,
83      1798, 1810, 1821, 1832, 1843, 1853, 1863, 1873,
84      1883, 1892, 1901, 1910, 1919, 1927, 1935, 1942,
85      1949, 1956, 1963, 1970, 1976, 1981, 1987, 1992,
86      1997, 2002, 2006, 2010, 2014, 2017, 2020, 2023,
87      2025, 2027, 2029, 2031, 2032, 2033, 2034, 2034,
88      2034, 2034, 2033, 2032, 2031, 2029, 2027, 2025,
89      2023, 2020, 2017, 2014, 2010, 2006, 2002, 1997,
90      1992, 1987, 1981, 1976, 1970, 1963, 1956, 1949,
91      1942, 1935, 1927, 1919, 1910, 1901, 1892, 1883,
92      1873, 1863, 1853, 1843, 1832, 1821, 1810, 1798,
93      1786, 1774, 1762, 1749, 1736, 1723, 1710, 1696,
94      1682, 1668, 1653, 1639, 1624, 1608, 1593, 1577,
95      1561, 1545, 1529, 1512, 1495, 1478, 1461, 1443,
96      1426, 1408, 1389, 1371, 1352, 1334, 1315, 1295,
97      1276, 1256, 1236, 1216, 1196, 1176, 1155, 1135,
98      1114, 1093, 1072, 1050, 1029, 1007,  985,  963,
99       941,  919,  896,  874,  851,  828,  805,  782,
100      759,  735,  712,  688,  665,  641,  617,  593,
```

```
101       569,  545,  521,  497,  472,  448,  423,  399,
102       374,  349,  325,  300,  275,  250,  225,  200,
103       175,  150,  125,  100,   75,   50,   25,    0
104 };
105
106   short half_sin2[HALF_SIN_LENGTH] = {
107        25,   50,   75,  100,  126,  151,  176,  201,
108       226,  251,  276,  301,  325,  350,  375,  400,
109       424,  449,  473,  498,  522,  546,  570,  595,
110       619,  642,  666,  690,  714,  737,  760,  784,
111       807,  830,  853,  876,  898,  921,  943,  965,
112       988, 1009, 1030, 1052, 1073, 1095, 1116, 1137,
113      1158, 1178, 1199, 1219, 1239, 1259, 1279, 1298,
114      1318, 1337, 1356, 1374, 1393, 1411, 1429, 1447,
115      1465, 1482, 1499, 1516, 1533, 1550, 1566, 1582,
116      1598, 1614, 1629, 1644, 1659, 1673, 1688, 1702,
117      1716, 1729, 1743, 1756, 1768, 1781, 1793, 1805,
118      1817, 1828, 1839, 1850, 1861, 1871, 1881, 1891,
119      1901, 1910, 1919, 1927, 1936, 1944, 1951, 1959,
120      1966, 1973, 1979, 1986, 1992, 1997, 2003, 2008,
121      2012, 2017, 2021, 2025, 2028, 2032, 2035, 2037,
122      2039, 2041, 2043, 2045, 2046, 2046, 2047, 2047,
123      2047, 2046, 2046, 2045, 2043, 2041, 2039, 2037,
124      2035, 2032, 2028, 2025, 2021, 2017, 2012, 2008,
125      2003, 1997, 1992, 1986, 1979, 1973, 1966, 1959,
126      1951, 1944, 1936, 1927, 1919, 1910, 1901, 1891,
127      1881, 1871, 1861, 1850, 1843, 1834, 1826, 1818,
128      1810, 1802, 1793, 1784, 1776, 1769, 1761, 1753,
129      1745, 1738, 1731, 1726, 1720, 1714, 1709, 1704,
130      1698, 1693, 1688, 1682, 1677, 1672, 1667, 1662,
131      1657, 1652, 1648, 1643, 1638, 1633, 1629, 1624,
132      1620, 1616, 1611, 1607, 1603, 1599, 1595, 1591,
133      1587, 1583, 1580, 1576, 1572, 1569, 1565, 1562,
134      1559, 1555, 1552, 1549, 1546, 1543, 1540, 1538,
135      1535, 1532, 1530, 1527, 1525, 1523, 1520, 1518,
136      1516, 1514, 1512, 1511, 1509, 1507, 1506, 1504,
137      1503, 1502, 1500, 1499, 1498, 1497, 1496, 1495,
138      1495, 1494, 1494, 1493, 1493, 1492, 1492, 1492
139   };
140
141
142   const unsigned char *in = (const unsigned char *) input_items[0];
143   short *out = (short *) output_items[0];
144   int ninput_items = (int)noutput_items/interpol_fac;
145
146   int step_lenght = (HALF_SIN_LENGTH * 2) / interpol_fac;
147
148   const short TABLE_LEVEL = 2047; // max absolute value of the signal
149   const float scaling = (float)(max - min) / (float)(TABLE_LEVEL * 2);
150   const float DC_offset = min + TABLE_LEVEL * scaling + 0.5; // round
151
152   // Verify that the interpolation factor is within range
153   assert (interpol_fac >= 4 && interpol_fac <= 512);
154   // Verify that the interpolation factor is dividable by 2
155   assert (interpol_fac % 2 == 0);
156   // and that (HALF_SIN_LENGTH * 2) / interpol_fac is a valid int
157   assert ((HALF_SIN_LENGTH * 2) % interpol_fac == 0);
158   // Verify that "sign" has a correct value
159   assert (sign == -1 || sign == 1);
160
161   for (int i = 0; i < ninput_items; i++)
162   {
```

```
163       if (in[i] == 0)
164       {
165         for (int j = 0; j < interpol_fac/2; j++)
166         {
167           out[interpol_fac * i + j] = (short) half_sin[step_lenght * j]
                * sign * scaling + DC_offset;
168           if (out[interpol_fac * i + j] < 0)
169             out[interpol_fac * i + j]--;
170         }
171         for (int j = interpol_fac/2; j < interpol_fac; j++)
172         {
173           out[interpol_fac * i + j] = (short) -half_sin[step_lenght * j
                - HALF_SIN_LENGTH] * sign * scaling + DC_offset;
174           if (out[interpol_fac * i + j] < 0)
175             out[interpol_fac * i + j]--;
176         }
177       }
178
179       else if (in[i] == 1)
180       {
181         for (int j = 0; j < interpol_fac/2; j++)
182         {
183           out[interpol_fac * i + j] = (short) half_sin2[step_lenght * j]
                 * sign * scaling + DC_offset;
184           if (out[interpol_fac * i + j] < 0)
185             out[interpol_fac * i + j]--;
186         }
187         for (int j = interpol_fac/2; j < interpol_fac; j++)
188         {
189           out[interpol_fac * i + j] = (short) half_sin2[HALF_SIN_LENGTH
                * 2 - step_lenght * j - 1] * sign * scaling + DC_offset;
190           if (out[interpol_fac * i + j] < 0)
191             out[interpol_fac * i + j]--;
192         }
193         sign = -sign;
194       }
195
196     }
197
198     return noutput_items;
199   }
```

Register

`dsrc_pulse_shaper_bs`, 42, 65
`gr-block`, 26
`m-block`, 26
`mpsk_receiver`, 45

ADC, *Se* analog-digital-omformer
Aktivitetsmonitor, 48
amplitudemodulasjon, 8
amplitudeskiftnøkling, 8
analog-digital-omformer, 16, 18
antialiasingfilter, 20
ASK, *Se* amplitudeskiftnøkling
AutoPASS, 33, 38

bærebølge, 6, 10, 12, 41
Babels tårn, 4
beregningskompleksitet, 31, 50
binær faseskiftnøkling, 6, 7
bitfeilrate, 7, 8, 34
BPSK, *Se* binær faseskiftnøkling

C++, 22, 29
CIC-filter, 14
CORBA, 28
CORDIC, 14
Costas-sløyfe, 12

DAC, *Se* digital-analog-omformer
DBPSK, *Se* differensiell binær faseskiftnøkling
dedikert kortholdslink, 33
  applikasjonslag, 38
  BST, 38, 48
  flagg, 37
  mediumtilgangskontroll, 38
  nedlink, 35
  opplink, 35
desimering, 14, 15, 45, 46
differensiell binær faseskiftnøkling, 7
digital-analog-omformer, 16, 18
DSRC, *Se* dedikert kortholdslink

Eclipse, 29
Extensible Markup Language, *Se* XML

faselåst sløyfe, 12
fasesynkronisering, 12
feilkorrigerende koding, 34
FIR-filter, 45
FPGA, 16, 18
frekvensbånd, 5

GNU Radio, 1, 22–26
  blokker, 24
  flytskjema, 22, 25
GNU Radio Companion, *Se* GRC
GRC, 25

halvbåndsfilter, 15

ikke-koherent deteksjon, 43
ikke-koherent mottaker, 46
inngangsimpedans, 20
interkomponentlatens, *Se* latens
interpolering, 14, 42

Joint Tactical Radio System, 5
JTRS, *Se* Joint Tactical Radio System, 26

koherent deteksjon, 43
koherent mottaker, 43
kommunikasjonssone, 36
kompleksitet, 31, 48, 50

latens, 31, 32, 39
linjekode, 11
  FM0, 11, 35
  NRZI, 11, 36
Linux, 22, 29, 40
logisk linkkontroll, 37

MAC, *Se* mediumtilgangskontroll
Mac OS X, 22, 29, 40
masteroppgave, 1, 51
MATLAB, 47
mediumtilgangskontroll, 19, 26, 36
mellomfrekvens, 14, 48
mikser, 12
minnebruk, 49, 50
mM&M, *Se* modifisert Mueller & Müller
modifisert Mueller & Müller, 13
modulasjon, 6–8, 10
mottaker, 6, 8, 12, 16, 18, 37, 43

nyquistraten, 42

OEF, 29
Open Source SCA Implementation:: Embedded, *Se* OSSIE
OSSIE, 29, 42, 49
OSSIE Eclipse Feature, *Se* OEF

programvaredefinert radio, 1, 4, 14
prosessorbelastning, 49, 50
Python, 22, 25

radioforsats, 20, 39, 40
referansesignal, 12

SCA, 26, 27
SDR, *Se* programvaredefinert radio
sender, 6, 7, 20, 37, 40
senderfilter, 11
Software Communications Architecture, *Se* SCA
støymargin, 43
symbolsynkronisering, 13
synkronisering, 12, 44

taktgjenvinning, 13

Ubuntu, 40
Universal Software Radio Peripheral, *Se* USRP
USB, 16, 19
USRP, 16, 29, 40
  datterkort, 20
  LFRX, 20, 43
  LFTX, 20, 40, 49
USRP2, 16

Windows, 22

XML, 25, 30

ytelse, 30, 31