FORORD

Denne rapporten beskriver vårt arbeid i den avsluttende oppgaven for vår utdanning ved Høgskolen i Bergen. Vi forutsetter at leserne har grunnleggene Linux- og programmeringskunnskaper.

Takk til Johan Alme, Ketil Røed, Bjarte Kileng, Håvard Helstrup, Matthias Richter, Dag Toppe Larsen, Dominik Fehlker og resten av IFT.

Bergen12. Juni 2006

Konvensjoner brukt i denne rapporten

Det følgende er en liste over konvensjoner som har blitt brukt i denne rapporten:

Kursiv

Blir brukt for fil og katalognavn, programmer og kommandoer, kommandolinjeopsjoner, email adresser og sti navn, web-adresser og alle nye uttrykk.

`Konstant bredde`

Blir brukt i eksempler for å vise innholdet av filer eller utskrift fra kommandoer, for å indikere systemvariabler eller nøkkelord som dukker opp i kode, og for maskinnavn.

`Konstant` **`bredde`** ***`fet`***

Blir brukt i eksempler for å vise kommandoer eller annen tekst som blir skrevet inn av bruker.

`Konstant bredde kursiv`

Blir brukt for å indikere variable opsjoner, nøkkelord eller tekst som en bruker skal bytte ut med en aktuell verdi.

Innholdsfortegnelse

FORORD......1

Konvensjoner brukt i denne rapporten......1

1.INNLEDNING......4

1.1.Oppdragsgiver......4

1.2.Problemstilling......5

1.2.1.A Large Ion Collider Experiment (ALICE)......5

1.2.2.Tidsprojeksjonskammeret (TPC)......5

1.2.3.Detector Control System (DCS)......7

1.2.4.Stråling......7

1.2.5.Vår oppgave......9

1.2.6.Oppsummering......10

1.3.Hovedidé for løsningsforslag......11

1.4.Prosjektform......12

2.DEFINISJON AV OPPGAVEN......13

2.1.Kravspesifikasjon......13

3.ANALYSE AV PROBLEMET......14

4.DESIGN AV MULIGE LØSNINGER......16

4.1.Plassering av funksjonalitet......16

4.2.Kommandolinjeverktøy......16

5.VURDERING OG VALG AV VERKTØY......17

5.1.Utviklingsverktøy......17

5.2.Annet......18

6.IMPLEMENTERING......19

6.1.Generelt om drivere......19

6.2.Kommunikasjon til enheten......20

6.3.Knytte sammen programvare og driver......24

6.3.1.Lese en ramme fra brukerprogram......24

6.4.Implementasjon i rcu-shellet......25

6.5.Proc-filsystemet (procfs)......25

6.5.1.Hva er procfs?......25

6.5.2.Hvordan det implementeres......26

6.6.Konfigurasjon av Virtex......29

6.7.Oppsummering......29

7.TESTING........30

7.1.Innledning........30

7.2.Selve testingen........30

8.BESKRIVELSE AV UTVIKLET SYSTEM I BRUK........32

8.1.Brukerdokumentasjon........32

8.1.1.Lesing av ramme........32

8.1.2.Lesing av register........32

8.1.3.Skriving til registre........33

8.1.4.Initiell konfigurasjon av Virtex enheten........34

8.2.Drifts- og vedlikeholdsdokumentasjon........34

9.OPPSUMMERING, DISKUSJON OG KONKLUSJONER........35

9.1.Avslutning........36

10.LITTERATURLISTE........37

11.VEDLEGG........38

1. INNLEDNING

1.1. Oppdragsgiver

Vår oppdragsgiver er Gruppen for eksperimentell kjernefysikk ved Institutt for Fysikk og Teknologi ved Universitetet i Bergen. Deres virksomhet er i hovedsak rettet mot akseleratoren *Large Hadron Collider (LHC)* og eksperimentet *ALICE*, som er under oppbygging på *CERN*, det europeiske laboratoriet for partikkelfysikk.

På den eksperimentelle siden bidrar de til konstruksjonen av to av de detektorer som skal samle data fra kollisjoner mellom blykjerner ved LHC. Det forventes at de innsamlede data vil gi svar på noen av de mest sentrale spørsmålene om universets utvikling og dets minste bestandsdeler, for eksempel hvorfor universet består av materie og ikke av antimaterie og hva som er opphavet til masse.

ALICE er et akronym for «A Large Ion Collider Experiment» og er et av de største eksperimentene innenfor forsking på egenskapene til materie i en svært liten skala. Prosjektet involverer en internasjonal kollaborasjon på mer enn 1000 fysikere, ingeniører og teknikere fordelt på 30 land.

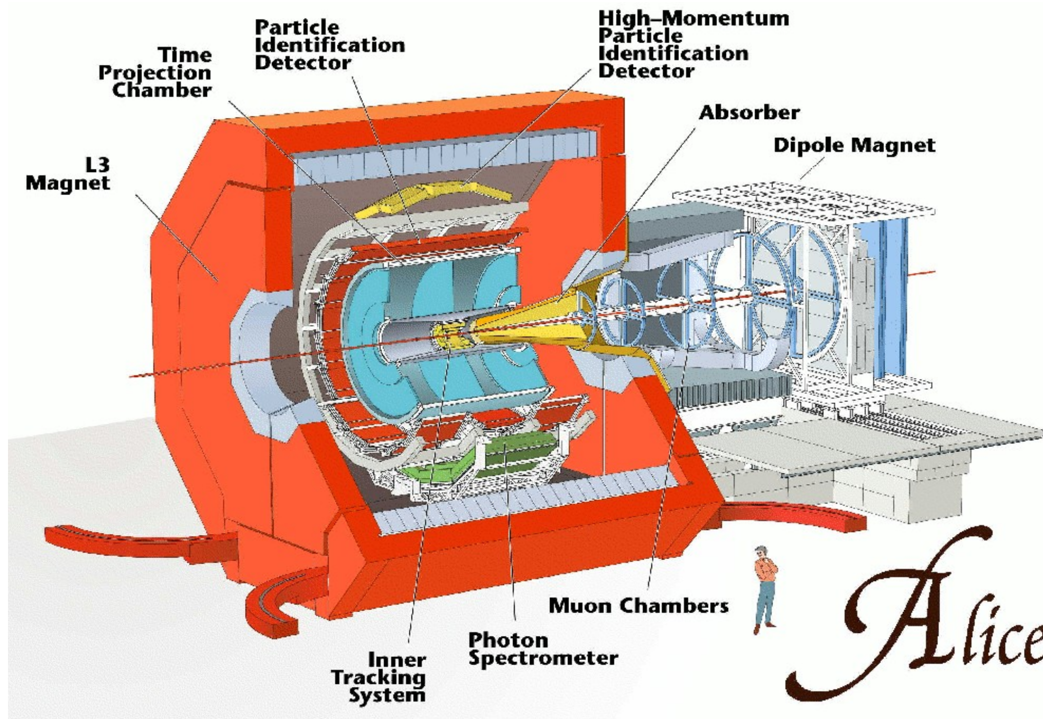
Eksperimentet vil generere enorme mengder data, i utgangspunktet mellom 50 og 100 GigaByte per sekund. Dette må reduseres ned til rundt 1-2 GB/s, som likevel er nok til å fylle 2 CD-ROM plater i sekundet.

Gruppen for eksperimentell kjernefysikk arbeider med en høynivå utløsermekanisme (*trigger*) som skal stå for en stor del av den nevnte datareduksjonen ved å velge ut hva som skal lagres for analyse. I tillegg har gruppen avnsvar for utforming av elektronikken som skal lese data fra tidsprojeksjonskammeret (*TPC*), og de kommer til å være aktiv i analysen av data i ettertid.

1.2. *Problemstilling*

Store deler av teksten er hentet fra følgende kilder: (Richter, 2005), (Xilinx UG012, 2005).

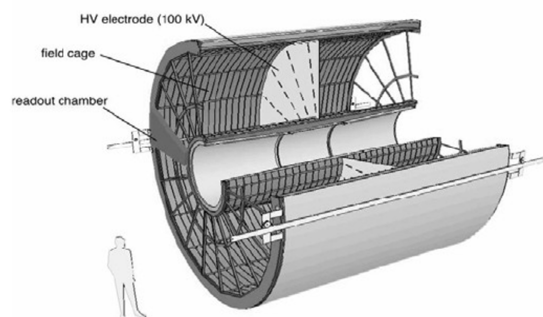
1.2.1. A Large Ion Collider Experiment (ALICE)



Illustrasjon 1: ALICE Detektoren

ALICE detektoren (Illustrasjon 1) skal samle inn data fra kollisjoner mellom blykjerner. Målet med ALICE er å generere høye nok energier til å lage og oppdage en materietilstand kalt *Kvark-Gluon plasma* (*Quark-gluon plasma/QGP*).

1.2.2. Tidsprojeksjonskammeret (TPC)

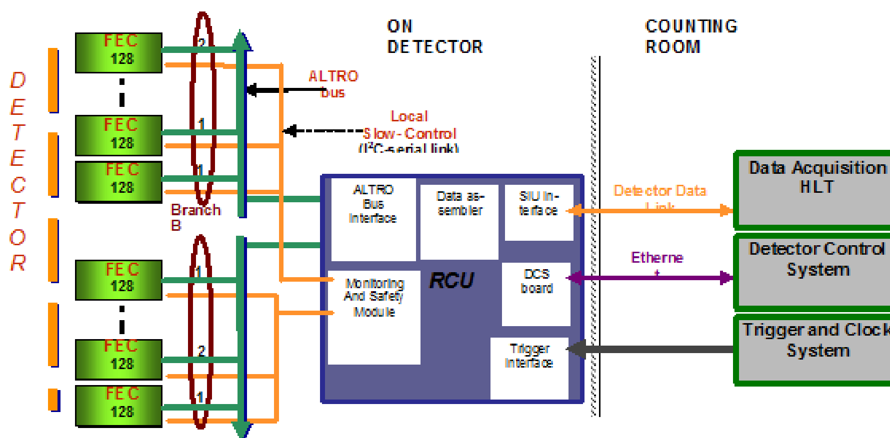


Illustrasjon 2: Tidsprojeksjonskammeret

Tidsprojeksjonskammeret (Illustrasjon 2) er en av hoveddetektorene i ALICE. TPC består 36 sektorer som totalt inneholder 4356 *Front-End Kort* (*Front-End Card/FEC*). Front-End kortene registerer hva som skjer under kollisjonene i TPC og filtrerer, prosesserer og digitaliserer disse dataene.

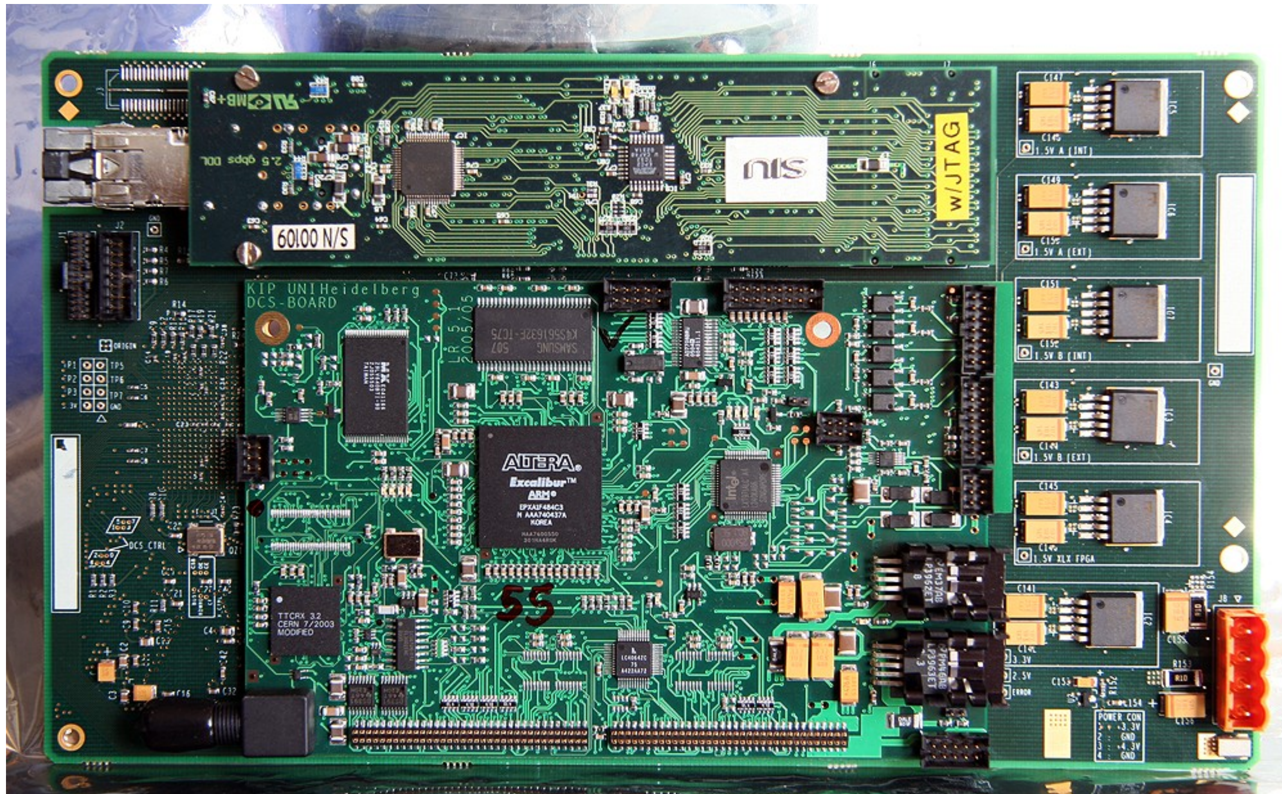
Hvert Front-End kort må konfigureres og overvåkes. TPC detektoren bruker en spesielt designet hardware enhet, en *Utlesnings-kontroll enhet* (*Readout Control Unit,* heretter kalt *RCU*), for å kontrollere Front-end kortene.

En enkel *node* i TPC Front-end elektronikken består av opp til 26 Front-end kort som er koblet til en RCU. En RCU består av et RCU hovedkort som er vert for to andre kort som også er spesielt designet for ALICE eksperimentet. *Detecor Data Link Source Interface Unit* (*DDL SIU*) er ALICEs standard grensesnitt til *Data Aquisition* (*DAQ*), hvor data fra eksperimentet blir sendt for lagring. Det andre kortet er *Detector Control System* (*DCS*) kortet som vi vil gå nærmere i detalj senere.



Illustrasjon 3: Skjematisk overblikk av Front-end elektronikken

Illustrasjon 3 er et skjematisk overblikk av Front-end elektronikken. Datastien går fra FEC via RCU til DAQ. Designet til RCU er basert på en *Field-Programmable Gate Array* (*FPGA*) som har mulighet for firmware oppgraderinger og nok plass til at man kan foreta andre operasjoner i tillegg til utlesingen av data.



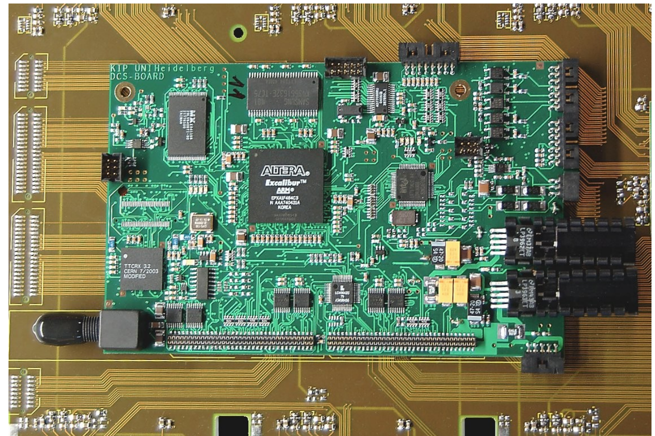
Illustrasjon 4: RCU hovedkort med DCS og SIU kort tilkoblet

En RCU, som vist i Illustrasjon 4, er basert på bruken av kommersielle hyllevarer som den nevnte FPGAen.

1.2.3. Detector Control System (DCS)

DCS er en av de store modulene i ALICE detektoren. Generelt vil dette systemet ta seg av oppgaver som kjøle- og ventilasjonssystem, magnetfelt og andre ting som konfigurasjon og overvåking av Front-end elektronikken. Dette systemet har blitt implementert på et eget kort, heretter kalt *DCS kort*.

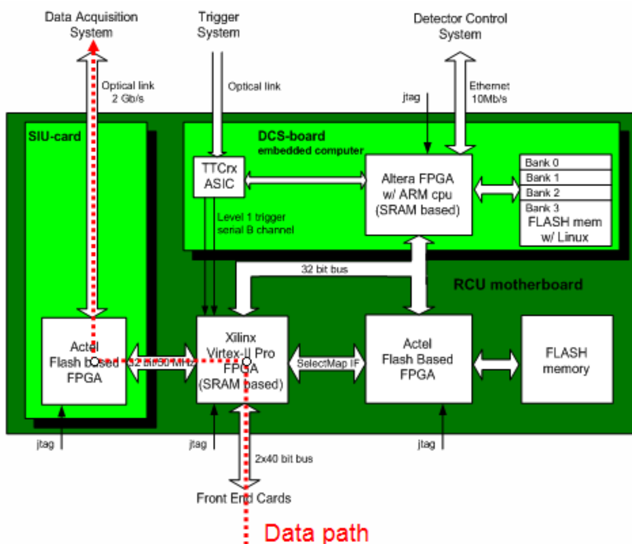
Selve DCS kortet (Illustrasjon 5) blir brukt i flere av subdetektorene i ALICE som en kontrollerende node på hardware nivået.



Illustrasjon 5: Detector Control System Kort

Kjernen i DCS kortet er en Altera FPGA som har en 32bit *ARM* prosessor med cache og en *Memory Management Unit* (*MMU*). I tillegg til FPGAen, inneholder kortet strålingstolerant *Flash ROM* på 8 MB, 32 MB *SDRAM*, et *Ethernet* grensesnitt, en analog til digital konvertor for volt og temperatur overvåking, en *JTAG* tilkobling og dedikerte datalinjer til RCU kortet. Disse komponentene gjør at DCS kortet er en skreddersydd, fullt fungerende datamaskin, som gjør det mulig å kjøre en lett-versjon av *Linux*. Ethernet grensesnittet gjør det også mulig å koble seg til DCS kortet utenifra, for eksempel gjennom en *SSH* tjener som kjører på kortet.

Registre og minne på RCU kortet er tilgjengelig for Linux på DCS kortet, enten direkte eller indirekte.



Illustrasjon 6: Skjematisk tegning av RCU med datastrømmen merket

DCS kortet skal operere uavhengig av hva som skjer på RCU kortet og i Front-end elektronikken. Man kan si at den skal leve sitt eget liv. Hovedoppgaven til DCS er å unngå at systemfeil vil avbryte datastrømmen fra detektoren til datalagringssystemet.

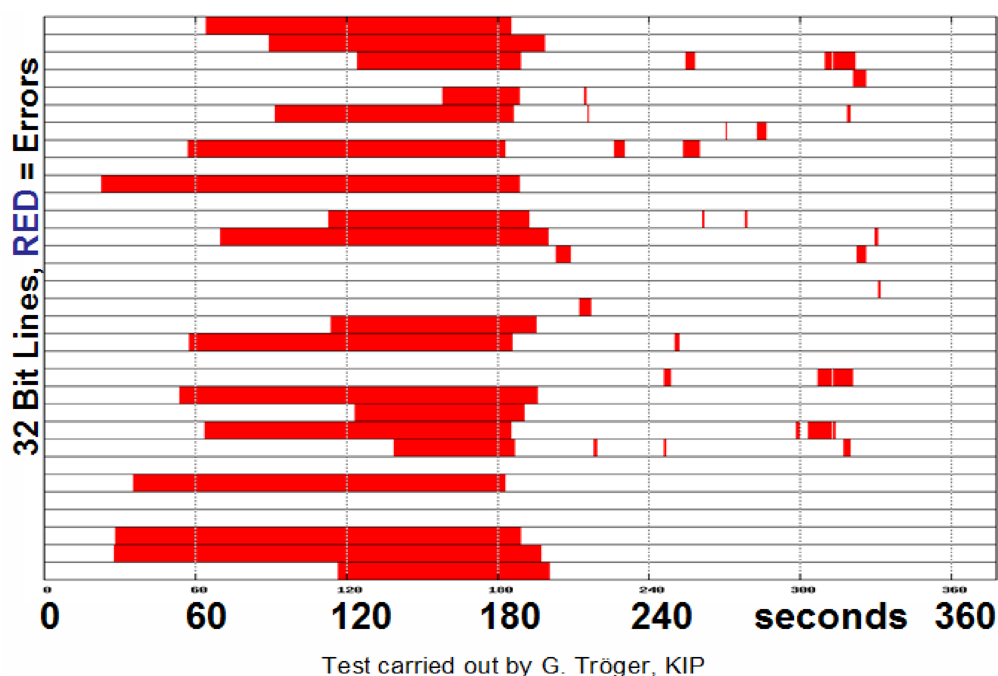
1.2.4. Stråling

ALICE detektoren jobber i et miljø med mye stråling. I et slikt miljø vil det kunne oppstå feil på Front-end elektronikken, og dette kan føre til funksjonsfeil i systemet. Det er viktig at slike feil blir oppdaget og opprettet så raskt som mulig, både for å forhindre permanent skade på utstyret og at datastrømmen blir avbrutt. Det er

ikke mulig å fysisk skjerme de sensitive delene fordi dette vil påvirke målingene. Hardware feil som kan ha oppstått kan heller ikke repareres uten å demontere hele detektoren.

Siden funksjonaliteten til både RCU kortet og DCS kortet er basert på SDRAM baserte FPGAer (se Illustrasjon 6), kan det oppstå feil i konfigurasjonsminnet på grunn av den høye strålingen i miljøet detektoren skal operere i. Disse feilene er ikke permanente, slik at de kan rettes opp ved å laste inn *firmware* på nytt. Firmware filene er lagret i strålingstolerante Flash ROM på de forskjellige kortene. Ulempen med å foreta en slik operasjon er at enheten som blir «oppfrisket» må startes på nytt, noe som vil føre til nedetid. For DCS kortet sin del er ikke dette kritisk, siden det opererer uavhengig av datastrømmen.

Denne fremgangsmåten vil ikke være tilstrekkelig for FPGAen på RCU kortet, siden dette fører til at datastrømmen blir avbrutt. Man har derfor basert systemet på en FPGA som tillater oppfrisking av firmware uten at operasjoner blir avbrutt. Denne teknikken kalles *Active Partial Reconfiguration (APR)*.



Illustrasjon 7: Feiltest med Xilinx Virtex-II Pro FPGA

I systemet har en Xilinx Virtex-II Pro FPGA (Heretter referert til som *Virtex*) blitt valgt. Arkitekturen til Virtexen er delt opp i rammer, som kan bestå av *RAM* elementer, *Configurable Logic Blocks* (*CLB*), *I/O* og så videre. Disse rammene er igjen delt opp i mindre elementer, kalt kolonner, som er de atomære delene i FPGAen. Når man gjør en APR adresserer man disse rammene og kolonnene ved å bruke rammeadresse og kolonneadresse, og slik er det mulig å gjenopprette en enkelt kolonne.

Illustrasjon 7 viser resultatet fra en funksjonell test av Virtex-II Pro enheten, gjort med et tregt grensesnitt slik at tidene er høyere enn de vil være under operasjon. De røde markeringer viser feil i minnet. Etter rundt 200 sekunder (i vanlig konfigurasjon tar dette noen millisekunder) blir hele firmware lastet inn på nytt. Vi kan se at det oppstår nye feil etterhvert, men disse blir kontinuerlig

rettet opp.

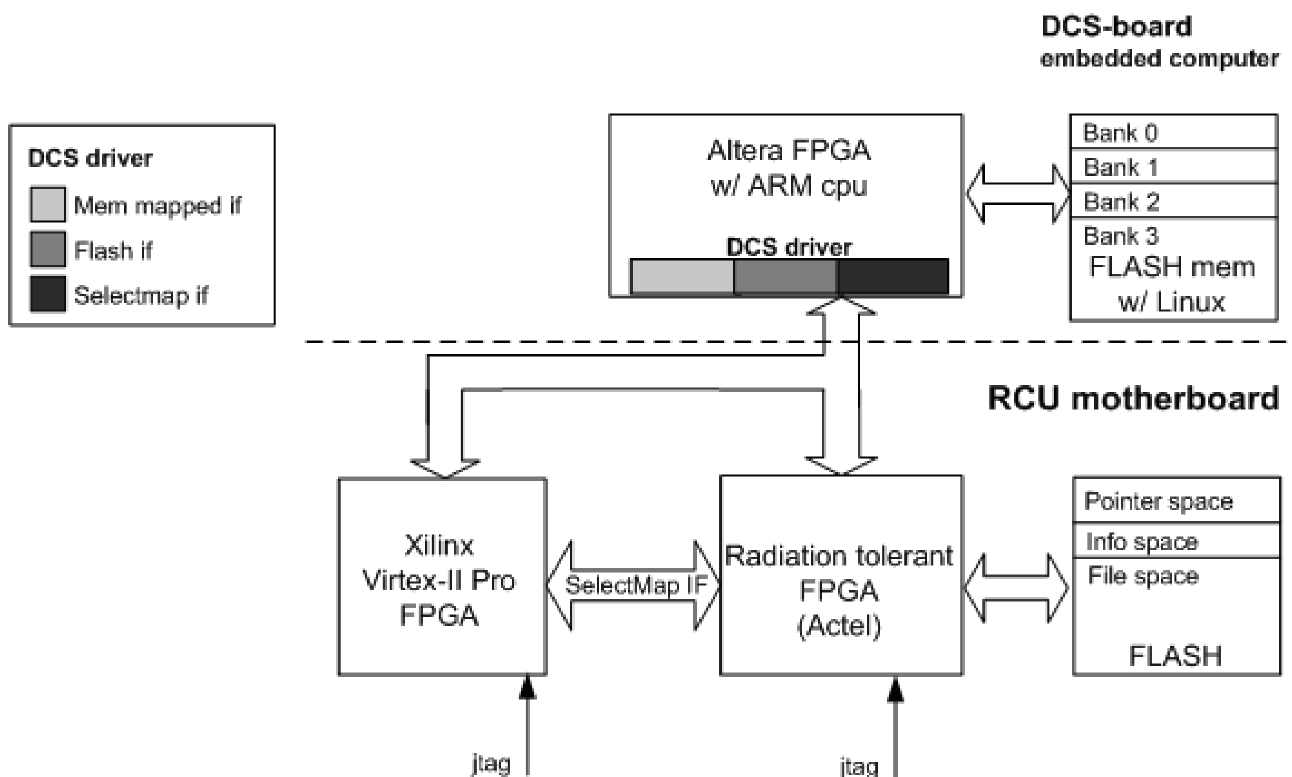
På RCU kortet vil konfigurasjon og rekonfigurasjon bli gjort ved å bruke *SelectMAP* buss grensesnittet, som er en 8 bit parallell buss. En komplett gjenoppretting av en Xilinx Virtex-II Pro VP7 vil ta ca 40 ms. For å kontrollere SelectMAP bussen og å kommunisere med det strålingstolerante Flash minnet er det benyttet en strålingstolerant FPGA fra Actel (Heretter kalt kun Actel) som har en spesielt designet firmware som vil gjøre det mulig å kontinuerlig friske opp konfigurasjonen på Virtex FPGAen (som vist i Illustrasjon 7), og hvis nødvendig, lese konfigurasjonsminnet og sjekke det mot den originale konfigurasjonen som er lagret på Flash minnet.

Så hvorfor ikke bare bruke Actel istedenfor Virtex? Problemet med Actel er at den ikke har noe mulighet for å forandre på konfigurasjonen uten at man har fysisk tilgang til den. Alle operasjoner er låst i firmware, og når den i tillegg er lokalisert 100 meter under bakken vil man ikke ha mulighet til å gjøre noe med den. Derfor er Actels oppgave kun å sjekke om konfigurasjonen på Virtex mot «fasiten» som er lagret på flash-minnet, og bytte den ut om det er nødvendig.

Ved å benytte en slik løsning er det mulig å «pause» lese-prosessen, rette opp feil i konfigurasjonen, og fortsette lesing når feilen er rettet. Dette oppsettet gjør det mulig å rette opp feil i konfigurasjonsminnet på Virtex FPGAen før de får innvirkning på resten av systemet. Firmwaren til Actel gjør også konfigurasjonsminnet tilgjengelig for DCS kortet, slik at firmware oppgraderinger er mulig eller en mer sofistikert feil-behandling.

1.2.5. Vår oppgave

Av alt dette er vår oppgave kun en liten del. Som nevnt så kjøres det en lettvektsversjon av Linux på DCS kortet, og DCS kortet har linjer for å kommunisere med Actel FPGAen. Som vi og har nevnt foregår konfigurasjon av Virtex gjennom et grensesnitt kalt SelectMAP, mer spesifikt via en protokoll som kalles Slave SelectMAP. For normale operasjoner vil Actel aksessere denne bussen (se Illustrasjon 8) for å lese og skrive konfigurasjonsdata til Virtex. SelectMAP grensesnittet er også gjort tilgjengelig for DCS kortet, gjennom Actel, hvor det er portet direkte til CPUen. Dermed kan SelectMAP grensesnittet bli kontrollert av en enhetsdriver i Linux som kjører på DCS kortet.



Illustrasjon 8: Skjematisk tegning av Xilinx rekonfigurasjons arkitekturen

Et slikt oppsett vil gjøre det mulig å implementere et høyt nivå av kontroll over konfigurasjonen på DCS kortet, og la Actel kun ta seg av de strålingskritiske oppgavene.

Siden forskjellige utgaver av DCS kortene også vil bli brukt på områder hvor stråling ikke er et tema vil det ikke være behov for en Actel FPGA på utlesningskontrollkortet (RCU). Der vil konfigurasjon av Virtex skje direkte fra DCS kortet.

Hardwaresiden av driveren aksesserer buslinjene direkte. Siden SelectMAP grensesnittet ikke har noen egne adresselinjer til å kommunisere med registerne, så må den interne adressen bli sendt over datalinjene i en bestemt rekkefølge.

Det eksisterer allerede en driver for SelectMAP grensesnittet, men denne er dårlig strukturert og har svært lite funksjonalitet; den kan kun laste inn en konfigurasjon til Virtex, ingenting annet. Selve SelectMAP grensesnittet er i tillegg kryptisk, så det er ønskelig med et enklere grensesnitt for kommunikasjon med det.

1.2.6. Oppsummering

Vi har i dette kapittelet nevnt veldig mye, kanskje for mye, om systemet. Men vi føler at det er nødvendig med et overblikk over systemet som vårt prosjekt skal være en del av. Uten en slik oversikt er det vanskelig å forstå problemstillingen.

For å oppsummere:

– RCU (utlesningskontroll) har som oppgave å ta imot, prosessere og sende videre data den får fra utlesningskortene (Front-end Card/FEC). Data sendes til et annet system, Data Aquisition.

- Utlesningskortene blir konfigurert og overvåket av en Virtex FPGA, som kan regnes som «hjernen» i RCU. I tillegg vil Virtex FPGAen utføre diverse prosessering av data fra FEC.
- På grunn av stråling må konfigurasjonsminnet på FPGAer kunne oppfriskes i tilfelle feil.
- Vi har tilgang til SelectMap grensesnittet fra DCS kortet, og har dermed muligheter for sofistikert feilbehandling og oppgradering av firmware.
- Nedetid er ikke aksepterbart i dataflyten, derfor er det valgt en FPGA som tillater oppfrisking av konfigurasjon uten at operasjoner avbrytes.
- Automatisk oppfrisking blir utført av en annen FPGA som er strålingstolerant. Denne FPGAen har ikke mulighet for oppgradering av firmware.

1.3. Hovedidé for løsningsforslag

Da oppdragsgiver presenterte oppgaven for oss hadde de noen forslag til hvordan det kunne gjøres. Likevel hadde vi ganske frie rammer fordi oppdragsgiver ikke hadde så mye erfaring med driverprogrammering selv. Hans forslag var å implementere atomære oppgaver som å skrive 32bits ord, lese 32bits ord, startup, init og abort. Disse atomære operasjonene kunne så kombineres til mer meningsfylte operasjoner gjennom software.

Softwaren skulle gi et mer brukervennlig grensesnitt til det mer kryptiske SelectMap interfacet, kunne lese/skrive til registre, og utføre de atomære oppgavene nevnt ovenfor. Det ble presentert to forslag på hvordan softwaren kunne implementeres. Enten kunne vi lage et helt nytt program fra grunnen av, eller vi kunne lage en tilleggsmodul til det eksisterende *RCU-shellet*. Vi vil forklare nærmere hva RCU-shellet er i kapittel 6.4 på side 25.

Et forslag for programmet var at den statiske delen av sekvensen med kommandoer (som trengs for å lese og skrive til et register), kunne bli lagret i konfigurasjonsfiler (muligens XML) slik at denne delen ville bli enkel å oppgradere hvis nødvendig.

Tilslutt skulle vi, hvis tiden strakk til, implementere ytterligere funksjonalitet, som foreksempel utlesing av konfigurasjonsminnet.

1.4. Prosjektform

Vi har valgt en prosjektform som er iterativ og vi har prøvd å jobbe etter Unified Process. Grunnen til dette er at vi liker den iterative prosjektformen ved å få ned risikoene så tidlig mulig i prosjektets tidsramme, og samtidig ha et kjørbart program. Likevel kan vi ikke si at vi har jobbet direkte etter Unified Process, for vi har for eksempel ingen klasser siden det ikke er støtte for klasser i porgrammeringsspråket som er brukt. På grunn av vår manglende innsikt i driverprogrammering har det vært vanskelig å kunne gjøre design i forkant av utvikling, siden vi har lært mens vi har utviklet.

Hver uke har vi hatt møte med arbeidsgiver hvor vi har diskutert løsninger, eventuelt om de tingene vi har gjort er de riktige i forhold til kravspesifikasjonen. På møtene er det også blitt diskutert hva som skal skje i forkant av neste møte. Vi har også snakket med oppdragsgiver og ressurspersoner på IFT når vi har hatt behov for dette. For eksempel når vi har sittet fast med et problem, eller vi har vært usikker på noe.

I forkant av forprosjektet lagde vi en fremdriftsplan som vi har klart å følge ganske godt.

2. DEFINISJON AV OPPGAVEN

2.1. Kravspesifikasjon

Vi føler det er viktig å presisere at til tross for at kravspesifikasjonen er relativt kort, så har arbeidet med de forskjellige funksjonene vært vanskelig. I tillegg så var denne kravspesifikasjonen kun et forslag fra arbeidsgiver på hvordan vi kunne gjøre det. De hadde selv ikke så mye erfaring med driverprogrammering og vi hadde som nevnt ganske frie rammer til å gjøre det på andre måter. Det viktigste var at driveren kunne lese og skrive til registre.

Vår oppgave var å utvikle et system bestående av:

1. En driver som kommuniserer med Xilinx Virtex-II Pro FPGA via SelectMAP interfacet.
   1.1 Implementere atomære instruksjoner som:
      - Skrive 32bits ord.
      - Lese 32bits ord.
      - Initialisere Virtex.
      - Starte opp Virtex.
      - Abort operasjon.

   1.2 Driveren skal være strukturert og lett å viderutvikle.
2. Et kommandolinje debuggingsverktøy som har tilgang til driveren.
   2.1. Verktøyet skal minst ha funksjoner som kan:
      - Skrive til et register.
      - Lese fra et register.

   2.2. Verktøyet kan utvikles fra grunnen av, eller lages som en tilleggsmodul til det eksisterende RCU-shellet (se s. 25).

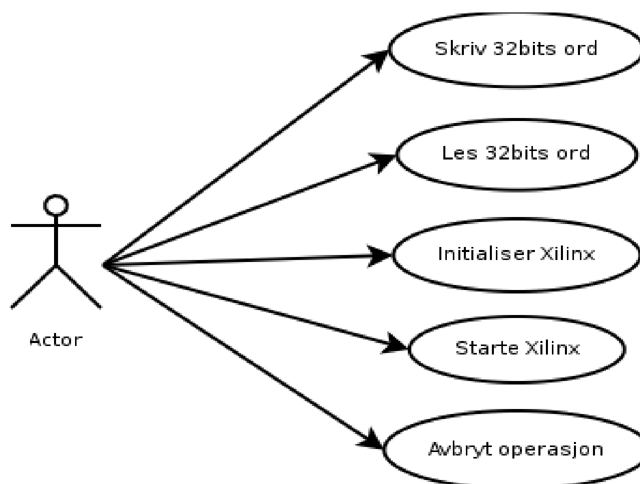
3. ANALYSE AV PROBLEMET

For å lage en driver i Linux må man har god kjennskap til hva en driver i Linux er, kunnskap om hvordan Linuxkjernen fungerer og inngående detaljkunnskap om enheten man skal skrive driveren for.

Da vi begynte på dette prosjektet hadde vi absolutt ingen erfaring med verken driverprogrammering eller hvordan Virtex fungerte. Det førte naturlig nok til at det var vanskelig å gjøre en grundig analyse i forkant av utviklingen. Dersom vi skulle ha noe håp om å bli ferdig i tide, måtte vi lære mens vi programmerte. Bare *databladet* til Virtex enheten var på over 500 sider (Hvorav rundt 100 sider var spesifikt om konfigurasjonen, og disse måtte vi kunne i detalj). Boken vi brukte, *Linux Device Drivers* (*Rubini et. al, 2001*), var og på over 500 sider og mesteparten av dette var relevant til oppgaven.

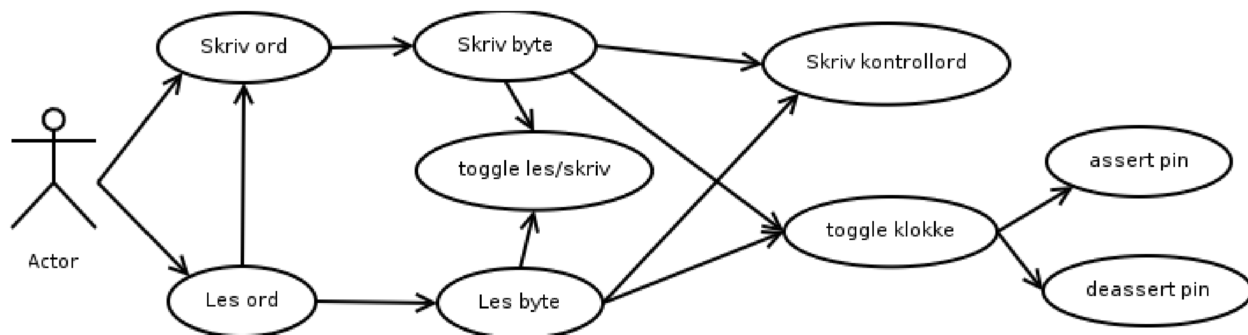
Den analysen vi gjorde i begynnelsen foregikk ved å studere kildekoden til den eksisterende driveren. Vi prøvde å forstå hvordan driveren gjorde ting, og hvordan vi kunne dele det opp i mindre operasjoner for å gjøre den mer strukturert. Utover dette prøvde vi å følge bokens *(Rubini et. al, 2001)* «mal» for oppbygging av en driver.

Vi lagde en foreløpig liste over operasjoner som vi trengte for å kunne implementere de atomære funksjonene (se Brukstilfelle 1) i kravspesifikasjonen. Disse operasjonene ville da danne grunnlaget og være bestanddeler i alle andre operasjoner som skulle implementeres senere.

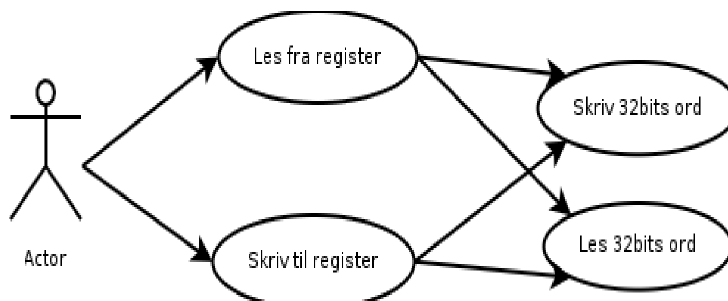


Brukstilfelle 1: Atomære funksjoner

For andre iterasjon var målet å kunne lese og skrive til et register, som vist i Brukstilfelle 3. For å få til dette trengte vi funksjonen for å kunne skrive 32-bits ord, som vist i Brukstilfelle 2.



Brukstilfelle 2: Lese og skrive ord(32bit).



Brukstilfelle 3: Lese/skrive til register.

For tredje iterasjon kunne vi velge å enten lage et helt nytt kommandolinje verktøy, eller å lage en tilleggsmodul til det eksisterende RCU-shellet. Valget falt ganske enkelt på RCU-shellet siden det ville være enklest for oss, og for dem som skulle bruke systemet. Det ville helt klart bli mye enklere for brukerne å forholde seg til et system fremfor to forskjellige.

4. DESIGN AV MULIGE LØSNINGER

På grunn av våre manglende erfaring med driverprogrammering var det vanskelig å lage design i forkant av utviklingen. Derfor vil denne delen av rapporten være relativt tynn, men vi vil likevel nevne de alternativene vi diskuterte underveis.

4.1. Plassering av funksjonalitet

Lesing og skriving til registre i Virtex foregår i en bestemt sekvens av kommandoer som sendes inn. Dette er spesifisert i manualen til Virtex. I hovedsak består disse sekvensene av 32bits ord som, et for et, enten leses eller skrives til Virtex.

En av tingene vi diskuterte i Iterasjon 2 var hvorvidt vi skulle implementere disse sekvensene av kommandoer direkte i driveren, eller om bare de mest elementene operasjonene som å lese et ord og skrive et ord skulle implementeres. I sistnevnte tilfelle ville det bety at et brukerprogram ville kunne lage meningsfylte operasjoner på egenhånd, som for eksempel lese registre.

For oss var det mest praktisk for oss å implementere alt i driveren fordi vi på dette tidspunkt ikke hadde fått implementert noe I/O kontroll (I/O kontroll forklares i kapittel på side). Mot slutten av prosjektet bestemte vi oss likevel for å tilrettelegge for at brukerprogram kunne lage disse operasjonene på egenhånd. Dette vil altså si at driveren vår nå har mulighet for begge deler.

4.2. Kommandolinjeverktøy

Som nevnt i kapittel 1.3, side 11, kunne vi velge om vi ville lage et helt nytt kommandolinjeverktøy, eller om vi skulle lage en tilleggsmodul til RCU-shellet (side 25). Å lage et helt nytt verktøy ville ha skapt masse ekstra arbeid for oss. I tillegg ville det neppe ha hatt noen fordeler kontra RCU-shellet, og brukerne av systemet hadde måttet forholde seg til to programmer, vårt og RCU-shellet.

Valget vårt falt på å lage en tilleggsmodul til RCU-shellet fordi vi følte det var den beste løsningen.

5. VURDERING OG VALG AV VERKTØY

For prosjektstyring brukte vi *Planner*, som er en prosjektstyrings applikasjon for *GNOME*. Vi synes Planner var lite intuitivt å bruke, men det hadde muligheten for å opprette Gantt diagram som vi benyttet oss av. Det gav oss i tillegg muligheten til å eksportere dette til *HTML* slik at det var enkelt å presentere det på prosjektsiden vår.

Dia er et *Open Source* program for tegning av strukturerte diagram. Dia ble brukt til å lage noen enkle brukstilfellediagrammer, men å lage sekvensdiagrammer ble ganske tungvint. Vi vurderte et program som heter *Umbrello*, men lot være å bruke det fordi det manglet endel funksjoner vi hadde behov for. Deretter prøvde vi et program som heter *Visual Paradigm*, og dette lot oss gjøre akkurat det vi ønsket. Visual Paradigm ga oss mye større frihet med tanke på hvordan vil ville lage diagrammene våre. Dessverre hadde vi bare tilgang til community-versjonen som er en nedstrippet versjon i forhold til fullversjonen, og lager dessuten et vannmerke på diagrammene. Uansett var det programmet som passet best for oss.

Vi valgte en webbasert løsning med *Wordpress* for dagboken/arbeidsloggen. Wordpress er i utgangspunktet laget for blogging, men dekket vårt behov på en tilfredsstillende måte. Imidlertid gikk dette skeis da en hyggelig sjel fant ut at han/hun skulle slette databasen hvor vi hadde lagret alle oppføringene, men dette hadde ikke noe å gjøre med Wordpress i seg selv.

5.1. Utviklingsverktøy

For å skrive selve koden trengte vi ikke noe mer enn en vanlig teksteditor. Når det gjelder UNIX systemer er det to editorer som er aktuelle, *Vi Improved* og *Emacs*. Andre editorer som *pico*, *nano*, *gEdit* o.l. har vi ikke vurdert i det hele tatt på grunn av deres manglende funksjoner. Siden vi har mest erfaring med «Vi» brukte vi denne.

All kompilering av koden har foregått med *GNU* verktøy og variasjoner av disse. For å få kode til å kjøre på DCS kortet må denne kompileres for ARM arkitektur. Vi har da brukt arm-uclibc pakken for dette. Alle filer som er inkludert i koden kommer fra ARM linux kernel kildekoden.

For versjonskontroll brukte vi *CVS* (*Concurrent Versions System*). CVS er et versjonskontrollsystem for software utvikling og data arkivering. Vi vurderte å bruke *SVN*(*Subversion*), men fordi ingen av oss har brukt dette før og fordi oppdragsgiver allerede brukte CVS var valget enkelt.

Til dokumentering av koden brukte vi *Doxygen*, som er noe lignende *Javadoc*. Doxygen generer automatisk dokumentasjon ut fra kildekoden i ulike formater som html og pdf. Doxgen ble også brukt av oppdragsgiver.

5.2. Annet

Vi vurderte å bruke *LaTeX* for alle tekstdokumenter som skulle produseres. Siden vi ikke har så mye erfaring med det, og heller ikke har så god tid til å sette oss inn i det, valgte vi å bruke Open Office.

Vi var såvidt borti bruk av *oscilloskop* for å sjekke at det vi leste/skrev til Virtex var korrekt.

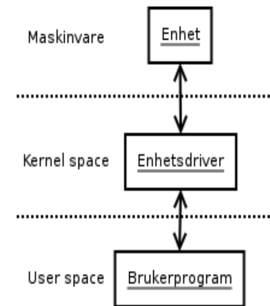
6. IMPLEMENTERING

6.1. Generelt om drivere

For å kunne kommunisere med maskinvare er vi avhengig av programvare for dette.

En enhetsdriver er en spesiell type programvare som er spesielt utviklet for å gi adgang til interaksjon med hardware enheter. Dette vil typisk gi et grensesnitt for kommunisering med hardware enhetene, gjennom den spesielle *bussen* som enheten er koblet til.

Med andre ord, en enhetsdriver, ofte forkortet til *driver*, er programvare som gir operativsystemet og annen programvare på høyere nivå midlene og metodene for å kontrollere og påvirke hardware enheter som er tilkoblet systemet Se forøvrig Illustrasjon 9.



Illustrasjon 9: Kommunikasjon mellom bruker og enhet

Driveren inneholder spesiell kunnskap om enheten og kan gi tilgang til enheten via veldefinerte grensesnitt. Dermed vil det for applikasjoner ikke være nødvendig å vite nøyaktig hvordan driveren fungerer eller hva som skjer på hardware nivået. Med en slik abstraksjon vil det også være enkelt å bytte ut driveren med en nyere versjon. Applikasjoner bruker de samme grensesnittene selv om driveren kanskje har implementert dem annerledes.

Dette vil si at bruksaktiviteter blir utført gjennom et sett av standardiserte kall som er uavhengig av den spesifikke driveren. Å kartlegge disse kallene til enhets-spesifikke operasjoner er ansvaret til driveren.

For Linux er programmeringsgrensesnittet slik at drivere kan bygges separat fra resten av kjernen og blir «plugget» inn når de trengs.

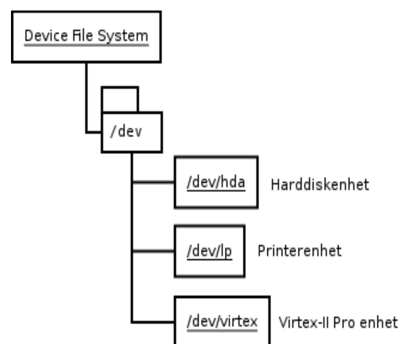
Generelt kan programmering deles i to deler. Hva funksjonalitet som skal stilles til rådighet, og hvordan denne funksjonaliteten skal brukes. Hvis disse to delene kan bli adressert av ulike deler av programmet, eller bedre, av ulike program, vil programvaren bli enklere å utvikle og å skreddersy til spesielle behov.

Når man skal utvikle en driver er det viktig å tenke på disse to delene. Driveren burde burde kun bry seg om å gjøre maskinvare tilgjengelig og overlate til applikasjoner hvordan maskinvaren skal brukes. Siden forskjellige miljøer vanligvis trenger å bruke hardware på forskjellige måter, er det viktig at driveren er så tilpasningsdyktig som mulig.

Å skrive en driver er regnet som en utfordring i de fleste tilfeller. Dette krever ikke bare en kunnskap om enheten man skal skrive driveren for, men også om maskinvaren den skal kobles til, og til operativsystemet som benyttes. Det er operativsystemet som gir rammene for hvordan enheten skal kunne kommuniseres med, og driveren må følge disse.

Det er selvsagt mulig å skrive programvare som tar direkte kontakt med maskinvaren, men dette vil ikke bli hverken fleksibelt eller særlig tilpasningsdyktig.

UNIX systemer er tungt basert på en filsystemstruktur, og omtrent alt i UNIX systemer kan bli behandlet som en fil. Dette gjelder også maskinvare, et eksempel er vist i Illustrasjon 10. I praksis vil man kommunisere med maskinvaren som man vil skrive til en fil. Det vil si at man kan sende enten en strøm av bytes til en enhet, eller en blokk av data. Disse enhetstypene kalles karakterenheter (*character devices*) og blokkenheter (*block devices*). Vi har også en tredje type, nettverksgrensesnitt, som vi ikke skal gå inn på her.



Illustrasjon 10: Enhetsfilsystemet

En typisk jobb for en driver er som regel å lese *I/O porter* og *I/O minne* (begge blir kalt *I/O regioner*). En *I/O port* er adressen til hvor enheten er koblet til, mens *I/O minne* er minneområdene som finnes på enheten. Dette vil si at man trenger å vite den nøyaktige adressen til hvor enheten er koblet til maskinvaren. F.eks så satt vi i flere timer uten å få noe respons fordi vi hadde gale hardwareadresser.

6.2. Kommunikasjon til enheten

Som nevnt tidligere må all kommunikasjon til SelectMAP grensesnittet foregå i et bestemt mønster. På Illustrasjon 11 kan man se en grafisk representasjon av denne kommunikasjonen. Her vises hvilke verdier i *kontrollordet* (forklart senere) som er 1 og 0.

Det vi gjorde først var å bryte ned mønsteret i spesifikke deler.

All konfigurasjon skjer ved å bruke et sett av *enhetspinner*. Disse pinnene kan enten være høy (1) eller lav (0). De forskjellige pinnene vi har tilgang til er følgende:

`CCLK` - Kontrollerer klokken.

`RDWR_B` - Bestemmer om SelectMap grensesnittet er satt for lesing (1) eller skriving (0).

`INIT_B` - Read-only, er høy hvis enheten er initialisert.

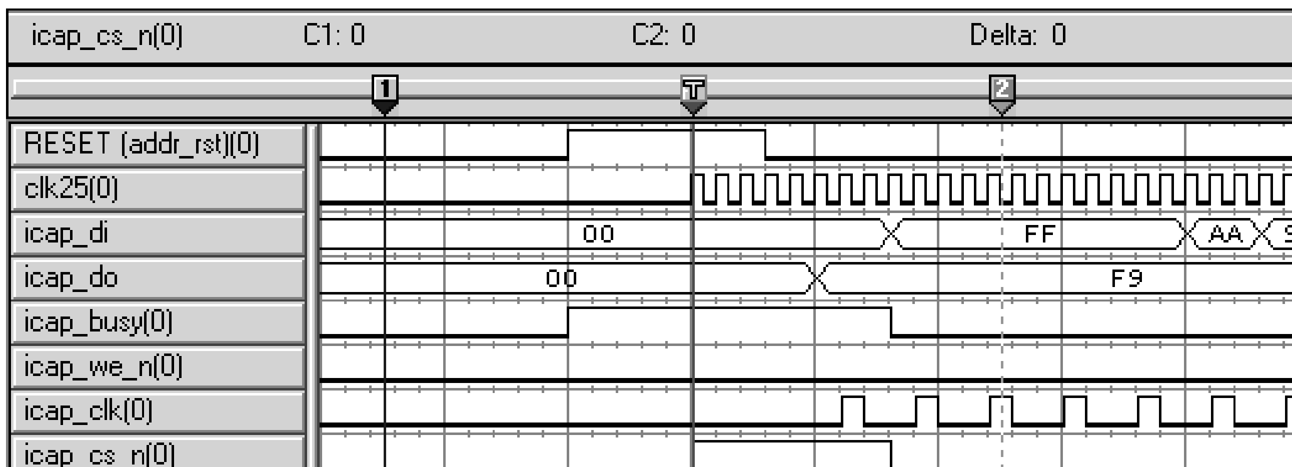
`PROG_B` - Full-chip reset, brukes til å slette all konfigurasjon.

`BUSY` - Denne er read-only og vil indikere at det har skjedd noe feil.

`CS_B` - slår av/på SelectMAP buss.

`DONE` - Read-only, hvis denne er høy har konfigurasjonen blitt lastet inn uten feil.

I tillegg var det en til, `BUS_SWITCH` som ble brukt internt på RCU for å bytte mellom *MemoryMap* og *SelectMAP* modus, men den har ingen funksjon lenger.



Illustrasjon 11: En grafisk representasjon av buss-linjene på SelectMAP grensesnittet

Alle disse pinnene samles i en byte, et *kontrollord*, som kan skrives og leses til en spesiell adresse. Det betyr at hver pinne har sin egen spesifikke plass. Dermed, hvis vi vil sjekke om enheten er initialisert kan vi se om den 2. bit'en er 1. Illustrasjon 12 viser hvordan kontrollordet er satt opp.

bit #	7	6	5	4	3	2	1	0
pin	BUS SWITCH	DONE	CS B	BUSY	PROG B	INIT B	RDWR B	CCLK

Illustrasjon 12: Kontrollordet for SelectMAP grensesnitt

Vi kan dermed bruke bitoperasjoner for å slå på/av de forskjellige pinnene. For eksempel for å toggle klokken kan vi lese kontrollordet, utføre en *OR* (eller) bitoperasjon slik at den 0. biten blir 1 og skrive kontrollordet tilbake. `CCLK` vil da gå fra 0 til 1.

Det første vi gjorde var å implementere funksjoner for å ta seg av all skriving av kontrollord og *asserting*(sette lav)/*deasserting*(sette høy) av pinner. De sistnevnte funksjonene vil da først lese kontrollordet og se om den spesifikke pinnen allerede er slik den skal være, og sette den hvis ikke.

For skifting mellom lesing og skriving kreves det annen funksjonalitet. Grunnen til dette er at SelectMAP grensesnittet kun har 8 pinner til data og bruker de samme for både lesing og skriving. Vi må derfor «snu» bussen hvis vi skal gå fra lesing til skriving. Dermed lagde vi en egen funksjon for dette.

Som nevnt er det kun 8 datapinner, det vil si at vi kun kan lese og skrive en byte om gangen. For lesing vil det bli sendt ut en ny byte hver gang det kommer en stigende klokke. Det vil si at vi må toggle klokken for hver byte som skal leses. Tilsvarende for skriving. I tillegg vil Virtex enheten legge/motta data i «feil» rekkefølge, slik at vi må reversere byten som skal leses/skrives.

Dermed har vi våre første «atomære» funksjoner, lese ett 32bits ord og skrive et 32bits ord.

Pseudokode for å lese ett 32 bits ord:

```
    Sett kontrollord til lesing
    for hver byte i ordet
        les byte fra datalinjene
        reverser byte
        sett byte inn i resultatordet hva bitoperasjoner
        toggle klokke
    returner resultatord
```

Tilsvarende for skriving.

Ved hjelp av disse funksjonene kan vi få til noe nyttig, nemlig å lese fra et register. For en oversikt over hvilke registre som finnes på Virtex, se Illustrasjon 13.

Register Name	Read	Write	Address	Description
CRC	Y	Y	00000	CRC register
FAR	Y	Y	00001	Frame address register
FDRI	N	Y	00010	Frame data input register (write configuration data)
FDRO	Y	N	00011	Frame data output register (readback configuration data)
CMD	Y	Y	00100	Command register
CTL	Y	Y	00101	Control register
MASK	Y	Y	00110	Masking register for CTL
STAT	Y	N	00111	Status register
LOUT	N	Y	01000	Legacy output register (DOUT for daisy chain)
COR	Y	Y	01001	Configuration option register
MFWR	N	Y	01010	Multiple frame write register
FLR	Y	Y	01011	Frame length register
KEY	N	Y	01100	Initial key address register
CBC	N	Y	01101	Initial CBC value register
IDCODE	Y	Y	01110	Device ID register

Illustrasjon 13: Oversikt over registre på en Xilinx Virtex-II Pro FPGA

En Virtex-II *bitstrøm* består av et 32-bits *synkroniseringsord* (`SYNC`) og et antall datapakker. Meningen med synkroniseringsordet er å «justere» konfigurasjonslogikken med begynnelsen på den første pakken i bitstrømmen. Hver pakke skal inn i et spesifikt konfigurasjonsregister for å sette konfigurasjonsopsjoner, programmere konfigurasjonsminnet eller å toggle interne signaler.

Bitstrøm pakkene består av en 32-bits header og en body av variabel lengde. Vi har to forskjellige typer headere, *Type 1,* som blir brukt for pakker opp til 2^{11}-1 ord, og *Type 2* som blir brukt til pakker på opp til 2^{27}-1 ord.

Hver av headerne spesifiserer hvilken type de er, om man skal lese eller skrive og antall ord. En Type 1 packetheader vil i tillegg kunne spesifisere en registeradresse. Dermed, hvis man skal skrive/lese flere ord enn 2^{11}-1 må man først sende en Type 1 header med register adressen, deretter en Type 2 header med antall ord.

Dersom vi skal lese ett ord fra et register må vi først sende inn sync ordet, deretter en Type 1 packet header for å spesifisere register adressen og hvor mange ord som skal leses. Etter denne må vi i tillegg «dytte» headeren inn i *bufferet*. Dette er på grunn av at alle pakkehoder passerer gjennom et 64-bits buffer før det når pakkebehandleren. For å «dytte» kommandoen videre må vi sende inn 64 bit til med data. Da har vi et spesielt ord som kalles `NOOP` (NO Operation) som vi sender to ganger.

Etter dette kan vi lese ordet. Etter at ordet har blitt lest må vi *desynche* enheten. Dette må også dyttes inn til pakkebehandleren ved hjelp av `NOOP`.

Dermed vil pseudokode for en les register funksjon se slik ut:

```
Sett kontrollord for skriving
Skriv SYNC
Skriv Type 1 Packet header med registeradresse og antall ord
Skriv 2 x NOOP
Sett kontrollord for lesing
resultat = Les 32-bit ord()
Sett kontrollord for skriving
Skriv DESYNCH
Skriv 2 x NOOP
```

En steg-for-steg gjennomgang av lesing av statusregisteret fra (Xilinx UG1012, 2005) kan sees i Illustrasjon 14.

Step	SelectMAP Port Direction	Configuration Data (hex)	Explanation
1	Write	AA995566	Sync Word
2	Write	2800E001	Read 1 word from STAT register
3	Write	20000000	NOOP
		20000000	NOOP
4	Read	SSSSSSSS	Device writes 1 word from the STAT register to the configuration interface
5	Write	0000000D	Desync command
6	Write	20000000	NOOP
		20000000	NOOP

Illustrasjon 14: Status Register Readback Sequence (SelectMAP)

For skriving til et register er det omtrent tilsvarende, men da må data skrives til enheten. Se også vedlegg for sekvensdiagram.

6.3. Knytte sammen programvare og driver

Som nevnt tidligere vil man kommunisere med drivere for maskinvare gjennom enhetsfiler. Disse må følgelig adresseres slik at det ikke er noen tvil om hva enhet man kommuniserer med. I UNIX systemer vil man adressere enheter ved hjelp av *major og minor* nummer. Major nummeret identifiserer driveren som er assosiert med enheten. Kernelen vil bruke major nummeret for å kalle opp videresende eksekvering til den tilhørende driveren. Minor nummeret blir bare brukt internt i driveren.

De enkleste filoperasjonene man kan gjøre med en enhetsfil er standard filoperasjoner som åpning av fil, lesing og skriving og lukking av fil. Driveren må da ha implementert disse funksjonene og hva som skal skje når en bruker åpner enhetsfilen, i vårt tilfelle `/dev/virtex`. Når denne filen åpnes vil funksjonen `virtex_open()` bli kalt, hvor vi foretar en initialisering av kontroll linjene og øker brukstelleren til driveren. Brukstelleren er her svært viktig, siden Virtexenheten kun kan brukes av en prosess om gangen. I klartekst vil dette si at andre brukere/prosesser må vente til enheten er helt ledig. Når man lukker filen, kalles funksjonen `virtex_release()`, vil brukstelleren minskes.

For å utføre mer spesifikke operasjoner utover filoperasjonene nevnt i forrige avsnitt må vi implementere noe som kalles I/O kontroll (Heretter kalt *ioctl*, siden det er slik det blir referert til i kernelen). I/O kontroll lar deg f.eks sende med argumenter som en tabell eller en datastruktur.

For å kunne implementere I/O kontroll trenger man et nummer, kalt «magic number», for å kunne identifisere ioctl kallene. Disse må være unike på systemet driveren skal kjøres på, for å forhindre krasj med andre drivere. For eksempel hvis to drivere bruker samme magic nummer vil ikke kernelen vite hvilken driver som skal håndtere kallet.

Når et brukerprogram kaller opp en I/O kontroll funksjon kan det f.eks være:

```
ioctl(fd, VIRTEX_READ_REG, header);
```

Dette kallet sendes fra brukerprogrammet og fanges opp av `ioctl()` funksjonen i driveren. Parameteren `VIRTEX_READ_REG` vil spesifisere hvilken oppgave I/O kontrollen skal utføre (lese et register), og er definert i switch-blokken i driverens `ioctl()` funksjon. Parameteren «header» inneholder data driveren trenger for å utføre denne oppgaven. Driveren vil da kalle opp sin funksjon `read_reg()` og lese en ramme hvor «header» vil spesifisere adressen det skal leses fra. Data som blir lest ut kan brukerprogrammet hente ut fra den samme variabelen den brukte til å sende data inn, nemlig «header».

6.3.1. Lese en ramme fra brukerprogram

Når et brukerprogram skal lese ut en ramme fra Virtex må det først spesifiseres en Type 2 header som angir antall ord som skal leses ut fra rammen. I tilegg til type 2 headeren må man oppgi en *FAC* (*Frame Address Composition*) header som spesifiserer hvilken ramme vi skal lese.

En ramme befinner seg i konfigurasjonsminnet til Virtex og består av 106 (32bits) ord. For å lese ut

en ramme må vi alltid først skrive ut 106 ord fra Virtex som kalles *padframe*. Padframe inneholder ikke noe nyttig data og forkastes. Et brukerprogram må derfor lese ut totalt 212 ord når det skal lese ut en ramme, men må forkaste de første 106 ordene.

Funksjonen for å lese ut en ramme i brukerprogrammet gjør to kall til `ioctl()` funksjonen i driveren. Først et kall som spesifiserer hvilke ramme vi ønsker å lese ut, og deretter et kall som spesifiserer antall ord som skal leses ut. `ioctl()` funksjonen i driveren kaller igjen opp funksjonen `read_frames()`, med Type 2 og FAC headeren som er blitt sendt med fra brukerprogrammet.

`ioctl()` funksjonen sender så en tabell tilbake til brukerprogrammet, og inneholder alle ordene som er blitt lest ut av `read_frames()`. Dette gjøres via en `copy_to_user()` funksjon fra driveren som er en standardfunksjon i driverprogrammering. Denne brukes til å kopiere data fra *kernel space* til *user space* (som er to adskilte minneområder). Dataene som er sendt tilbake skrives så til fil i brukerprogrammet, men kan også presenteres via utskrift på skjermen.

6.4. Implementasjon i rcu- shellet

RCU-shellet (`rcu-sh`) er et debuggingsverktøy og benytter seg av forskjellige operasjonsmoduser for å utføre diverse oppgaver på RCU. Skal man f.eks gjøre operasjoner mot flashminnet må man gå inn i *flashmodus*. Tilsvarende er det for *memorymapped modus* og *SelectMAP modus*. RCU-shellet er bygget opp slik at det er moduliserbart. Når man går inn i en bestemt modus vil tilhørende funksjoner, som kan ligge i en modul, kalles opp.

Driveren vår benytter seg av SelectMAP grensesnittet, og når SelectMap modus blir slått på vil funksjonen `initSmAccess()` bli kalt opp. Denne funksjonen foretar de nødvendige initialiseringene som trengs for å få tilgang til driveren. På et UNIX-system vil dette si at man åpner filen som driveren er knyttet til.

Når initialiseringen er gjort kan en bruker via RCU-shellet f.eks skrive *`sm rr 14`* som vil si at man vil lese register 14 (`IDCODE` registeret, se også Illustrasjon 13 på side 22). Dette vil kalle opp funksjonen `smRegisterRead()`, som ved bruk av en hjelpefunksjon `smMakeT1Header()` tar tallet 14 og lager en Type 1 header som bestemmer hvilket register det skal leses fra. Programmet gjør så et `ioctl()` kall som har samme fremgangsmåte som vi har beskrevet på side

Når registeret er lest vil innholdet bli sendt tilbake til RCU-shellet og vises for brukeren.

I Skjermdump 1 på side 32 viser vi hvordan man starter `rcu-sh` og resultatet av kommandoer..

6.5. Proc- filsystemet (procfs)

6.5.1. Hva er procfs?

På Unix systemer er *procfs* forkortet fra *Process filesystem*. Prosess filsystemet er et spesielt programvare-filsystem (et pseudo-filsystem) som blir brukt av kernelen for å kunne eksportere informasjon. Hver fil under `/proc` er knyttet til en kernel funksjon som genererer filens «innhold» når filen blir lest.

/proc blir brukt mye i Linux systemer. Programmer som ps, top og uptime henter sin informasjon fra. /proc filsystemet er dynamisk, slik at kernel moduler kan legge til og fjerne oppføringer når som helst.

Å lage en fullblods /proc oppføring kan være komplisert, siden de også har mulighet for å kunne skrive til dem. Vi har kun laget oppføringer som kan leses fra.

6.5.2. Hvordan det implementeres

Som nevnt, må det for hver oppføring under /proc knyttes en funksjon. Kernelen har definert et grensesnitt for hvordan slike funksjoner skal se ut. Det finnes faktisk to slike grensesnitt, men det er anbefalt at man bruker det som kalles read_proc(). Det andre som finnes, get_info(), er et eldre grensesnitt.

En funksjon for å lese proc oppføring vil da se slik ut:

```
int (*read_proc) (char *page, char **start, off_t offset,
                  int count, int *eof, void *data)
```

En kort oppsummering av noen argumentene: *page* pekeren er posisjonen til bufferet hvor data skal skrives til; *eof* peker til en integer som må settes av driveren for å signalisere at den ikke har noe mer data å returnere. *data* er en driverspesifikk data peker som brukes til intern «bokføring».

For å kunne lage en /proc oppføring må man opprette en struktur som kalles proc_dir_entry. Denne strukturen inneholder informasjon om navnet til oppføringen(f.eks *info*), lokasjonen til den (f.eks /driver/virtex/), hvilke funksjoner som skal knyttes til lesing og skriving av oppføringen og tilslutt en *data* struktur.

Siden vi har brukt *data* strukturen mye for vår implementasjon må vi nesten fortelle litt om den, men først litt om hva som er vår tankegang for /proc oppføringene.

Vi har brukt procfs som en metode for å hente ut informasjon om registrene på virtex enheten. På enheten har vi 15 forskjellige registre. Hvordan man kan aksessere disse registrene er forskjellig, noen kan kun leses fra eller kun skrives til, mens noen har både lese og skrive tilgang. Hvert register har sitt unike navn og adresse. Hvert register har sin egen /proc oppføring, i tillegg har vi en *info* oppføring som vil liste ut informasjon fra flere registre som kan være aktuelle å se på.

Siden vi har 15 registre som stort sett er ganske like ville det vært ganske upraktisk å lage 15 (egentlig 10, siden det kun er 10 registre man kan lese fra) funksjoner som mer eller mindre gjør det samme.

Med *data* pekeren som er nevnt tidligere er det mulig å implementere ett enkelt kall for alle registrene. Måten registrene blir aksessert på er lik, den eneste forskjellen er navnet på registeret og adressen til det. Dermed kan vi lage en struktur som inneholder navnet og adressen til hvert register, og hvilke operasjoner (lese/skrive) man kan utføre på registeret. Den endelige proc oppføringsstrukturen vår inneholder to strukturer, en for *data* strukturen og en for selve proc oppføringen. Med en slik oppdeling er det enkelt å legge disse inn i en tabell slik at vi kun trenger å iterere oss gjennom den for å lage de enkelte oppføringene. Vi vil nå gå nærmere i detalj på

hvordan dette er implementert.

For selve datastrukturen er det to felter. Ett for navnet på registeret og ett for hvilken adresse det har. Selve strukturen ser slik ut:

```
struct reg_data_t {
    char name[REGNAME_LEN+1];
    unsigned short int address;
};
```

Dette pakkes inn i nok en `struct`, men nå legger vi til informasjon om aksesseringsmulighetene og selve `proc` filinformasjonen.

```
typedef struct {
    struct reg_data_t data;
    unsigned char read;
    unsigned char write;
    struct proc_dir_entry *file;
} entry_t;
```

Vi har i tillegg laget dette som en egen type, `entry_t`, slik at vi enkelt kan opprette `proc` oppføringer.

Det neste vi må gjøre er å legge disse inn i en tabell. Dette gjøres slik:

```
static entry_t proc_entry[] = {
//      NAME        ADDRESS         R  W  FILPTR
    { {"CRC",       CFGREG_CRC },   1, 1, NULL },
    ...
    { {"IDCODE",    CFGREG_IDCODE},1, 1, NULL }
};
```

For å enkelt aksessere de forskjellige elementene i tabellen lagde vi og en *enum*:

```
enum V_REGS { CRC, FAR, FDRI, FDRO, CMD, CTL,
              MASK, STAT, LOUT, COR, MFWR, FLR,
              KEY, CBC, IDCODE };
```

Selve opprettingen av `proc` oppføringer skjer i en egen funksjon, `virtex_create_procs()`,

som blir kalt opp fra initialiseringen av modulen, `virtex_init()`. Siden vi har lagt dem inn i en tabell kan dette gjøres ganske enkelt i en for løkke:

```
for (i = 0; i < MAX_REGS; i++) {
     proc_entry[i].file = create_proc_entry(
                                  proc_entry[i].data.name,
                                  mask,
                                  virtex_reg_dir);
}
```

Merk at dette er sterkt forenklet. *`mask`* er rettighetene til den spesifikke filen (`/proc` oppføringen) som blir opprettet. Den blir beregnet ut ifra verdiene til *`proc_entry[i].read`* og *`.write`*. *`virtex_reg_dir`* er en egen katalog under `/proc` (`/proc/driver/virtex`) hvor vi legger alle oppføringene.

I tillegg må vi knytte lese og skrive funksjonene til oppføringen, og «eieren» av filen. Eieren må spesifiseres, slik at kernelen kan holde orden på om driveren er i bruk eller ikke. Dette gjøres slik:

```
proc_entry[i].file->data = &proc_entry[i].data;
if (proc_entry[i].read)
     proc_entry[i].file->read_proc = proc_read_reg;
if (proc_entry[i].write)
     proc_entry[i].file->write_proc = proc_write_reg;
proc_entry[i].file->owner = THIS_MODULE;
```

Som vi ser har vi to forskjellige funksjoner som kan knyttes til oppføringen. En for lesing, `proc_read_reg()`, og en for skriving, `proc_write_reg()`. `proc_write_reg()` funksjonen returnerer bare `0` siden vi ikke har hatt tid til å implementere den.

Vi ser også at `file->data` vil peke til vår datastruktur for registrene, slik at hver oppføring vil ha sin egen data struktur.

Siden driveren vår er laget som en modul vil det si at den kan legges inn og fjernes fra kernelen når som helst. Derfor er det viktig at også `proc` oppføringene fjernes med dem. Igjen, med vår tabell implementasjon blir dette enkelt.

```
for (i = 0; i < MAX_REGS; i++)
     remove_proc_entry(proc_entry[i].data.name,
                       virtex_reg_dir);
```

Vi vil nå gå gjennom hva som skjer når man aksesserer en av våre `proc` oppføringer.

Selve funksjonskallet med argumenter ser slik ut:

```
static int proc_read_reg(char *page, char **start,
                         off_t off, int count, int *eof,
```

```
                    void *data)
```

Det første vi gjør etter noen driverspesifikke ting, er å hente ut data strukturen. Den ligger i pekeren `data`. Siden dette er en `void` peker kan den peke til hva som helst, så vi må kaste den til typen som vi trenger.

```
struct reg_data_t *reg_data = (struct reg_data_t *)data;
```

Deretter er det bare å bruke funksjonen vi har laget for å lese fra register, `read_reg()`, og hente ut det som ligger der. Resultatet legger vi i variabelen *`value`*. Vi må også returnere lengden av teksten vi skal gi tilbake, den har vi i variabelen *`len`*. Dette gjør vi enkelt slik:

```
len = sprintf(page, "%s register (address: %#.5x)"
                    " : %#.8lx\n",
                    reg_data->name,
                    reg_data->address, value);

return len;
```

6.6. Konfigurasjon av Virtex

Den originale driveren hadde funksjoner for å laste inn konfigurasjon av enheten. Denne har vi oppdatert til å bruke våre funksjoner og generelt skrevet om mesteparten av den eksisterende koden. Konfigurasjon må også utføres i et spesielt mønster. Blant annet vil vi på slutten av konfigurasjonen toggle klokken 30 ganger for å være helt sikker på at alt har kommet inn i enheten. Vi må og behandle konfigurasjonsfilen på en spesiell måte siden denne kan ha en egen header som ikke har noe med selve konfigurasjonen på Virtex å gjøre. Derfor vil funksjonen først se etter fire etterfølgende `SYNC` og lese konfigurasjonen derfra.

6.7. Oppsummering

Driveren vår har implementert følgende funksjonalitet:

- Lese og skrive 32-bit ord
- Lese og skrive til register
- Lese rammer i konfigurasjonsminnet
- Kall for å viderebringe informasjon fra enheten til brukerprogrammer, både via `procfs` og `ioctl`.
- Mulighet for en full rekonfigurasjon av enheten.

I tillegg har vi laget en modul for et eksisterende kommandolinje debuggingsverktøy som benytter seg av kallene som er implementert i driveren for å hente og skrive informasjon fra enheten.

7. TESTING

7.1. Innledning

Testing av en driver er et kapittel for seg selv. Vi har ikke den samme muligheten til å lage *Unit*-tester som man har med vanlige programmer. Vi kan heller ikke undersøke koden på samme måte som man ville gjort med en debugger.

Kernel kode kan ikke på en enkel måte bli kjørt under en debugger, og det kan heller ikke «spores» siden det er et sett av funksjonaliteter som ikke er relatert til noen spesifikke prosesser. Feil i kernel koden kan også være vanskelig å reprodusere og kan i verste fall ta ned hele systemet, og dermed ødelegge bevisene som kunne blitt brukt til å spore feilen.

Den mest vanlige måten å debugge på er ved direkte utskrift fra programmet. Denne muligheten har man også når man skal debugge kernel kode. Mens man i vanlige programmer f.eks ville brukt funksjonen `printf()`, har man i kernelen en egen funksjon som heter `printk()` (man kan heller ikke bruke standard *C* biblioteket fra kernelen, siden man da måtte ha brukt minne for det). En av forskjellene til `printk()` i forhold til `printf()` er at man kan klassifisere meldinger etter ulike *log-nivå*. Nivået `KERN_EMERG` brukes i kritiske situasjoner, for eksempel noe som vil føre til et system krasj. Vi har 8 forskjellige nivåer, rangert etter hvor kritiske de er. Det laveste, `KERN_DEBUG`, vil bli brukt for debuggings beskjeder.

Disse kernelbeskjedene blir loggført i `/var/log/messages` slik at vi kan følge med på hva som skjer når vi utfører operasjoner til driveren. Ulempen med en slik løsning er at hvis man har mye debug (for eksempel at man skriver ut hver byte i en strøm) vil ikke alltid programmet som skriver til log-filen greie å få med seg alt, slik at noen meldinger kanskje vil falle bort eller komme i feil rekkefølge. Merk at dette kun skjer i spesielle situasjoner; som oftest går det bra.

Det finnes også andre måter å debugge på, men siden vårt system kjører en svært forenklet versjon av linux har vi ikke tilgang til mange disse. Vi går derfor ikke nærmere inn på dem her. For vår del har debugging ved utskrift til log vært tilfredsstillende. Men det har selvfølgelig dukket opp situasjoner hvor vi har gjort noe og det eneste som har skjedd er at systemet har krasjet. For eksempel vil en evig loop ikke gjøre noe annet enn å henge systemet slik at vi må skru av strømmen på det. DCS kortene er designet for å være mest mulig *autonyme*. Dette vil si at etter en *reboot* skal de kunne fortsette å operere som normalt. En ting vi fant ut er at hvis man fyller opp minnet på dem, så vil de reboote av seg selv. Dette har spart oss et par turer ned til rommet der de står for å slå dem av og på.

7.2. Selve testingen

Vår plan for testing har vært å teste hele tiden mens vi har utviklet kode. For hver ny funksjonalitet eller brukstilfelle vi har implementert har vi gjort tester om driveren fungerte på systemet som driveren var laget for. Det har vært eneste måte vi virkelig kunne sjekke om ting var gjort riktig.

Testene vi har gjort har vært å skrive ut debuggingsmeldinger i *kernel-loggen*, utskrifter i testprogram eller avlesing av bitverdier med oscilloskop på DCS kortet. Vi har sjekket at registrene

vi har lest ut stemmer i forhold til manualen til Virtex enheten, og vi har sjekket at rammene vi har lest ut stemmer i forhold til rammer som oppdragsgiver har lest ut før med andre metoder (mer tungvinte metoder). Skriving til registre har blitt feilsjekket etter at lesefunksjonalitet har blitt verifisert, og metoden dette er blitt gjort på er at vi har lest av et register, skrevet til registeret, og lest av registeret igjen for å se om det vi skrev til registeret virkelig ble skrevet.

Vi har også laget et lite testprogram som tester ut driverens funksjonalitet og samtidig illustrerer hvordan driveren kan implementeres i software. Vi har også gjort tester på at driveren fungerer i rcu-sh.

Når feil er blitt oppdaget har gruppen som regel jobbet sammen for å utbedre feilen, og i enkelte tilfeller har vi måtte jobbe sammen med oppdragsgiver for å finne ut hva som var galt. Slik har vi også fått en forsikring om at vi var på rett spor da vi utviklet koden.

Driveren vi har laget har blitt kjørt kontinuerlig over noen dager. Noe lenger tidsrom har vi ikke hatt mulighet til å teste på grunn av prosjektets tidsramme. Under kjøringen har det ikke dukket opp noen problemer.

8. BESKRIVELSE AV UTVIKLET SYSTEM I BRUK

8.1. Brukerdokumentasjon

Systemet kan lese registre og rammer, samt skrive til registre. Vi forutsetter at bruk av RCU-shellet og enkle Linux kommandoer er kjent.

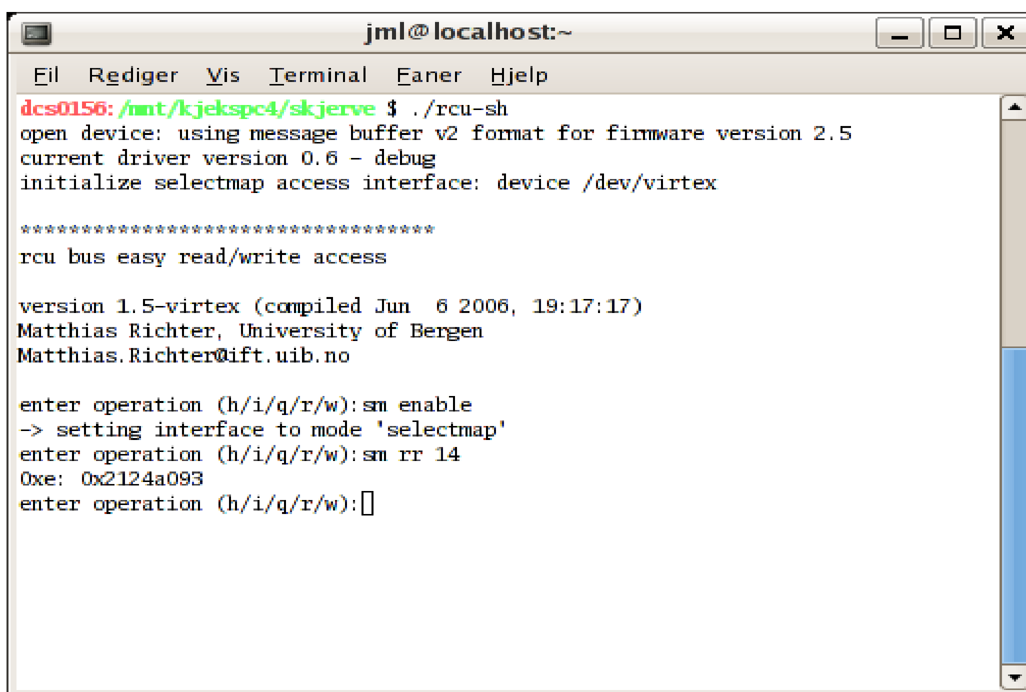
8.1.1. Lesing av ramme

Lesing av rammer kan foreløpig bare gjøres i testprogrammet vårt, som leser rammen og skriver den til fil. Kommando er:

```
./virtex_test <block address> <majornumber> <minornumber>
```

Parameterne spesifiserer hvilken ramme som skal leses ut, og krever at man har kjennskap til Xilinx Virtex-II Pro og hvordan konfigurasjonsminnet er oppbygd.

8.1.2. Lesing av register



Skjermdump 1: Bruk av RCU-shellet

Lesing av registre kan gjøres på to måter. Enten ved å bruke `rcu-sh` (for demonstrasjon se Skjermdump 1), eller ved å bruke proc filsystemet (for demonstrasjon se Skjermdump 2.

- For RCU-shellet

  For å lese et register må man spesifisere hvilket register man vil lese. Kommando er:

  ```
  sm rr <registeradresse>
  ```

  Registrene er adressert internt på Virtex enheten fra 0 til 14. Det er ikke mulig å lese fra alle registrene. Dette er spesifisert i manualen til Virtex enheten. Registeradressen må oppgis i desimaltall.

- For proc filsystemet

  For å lese ut registrene via proc må man gå inn i mappen `/proc/driver/virtex`

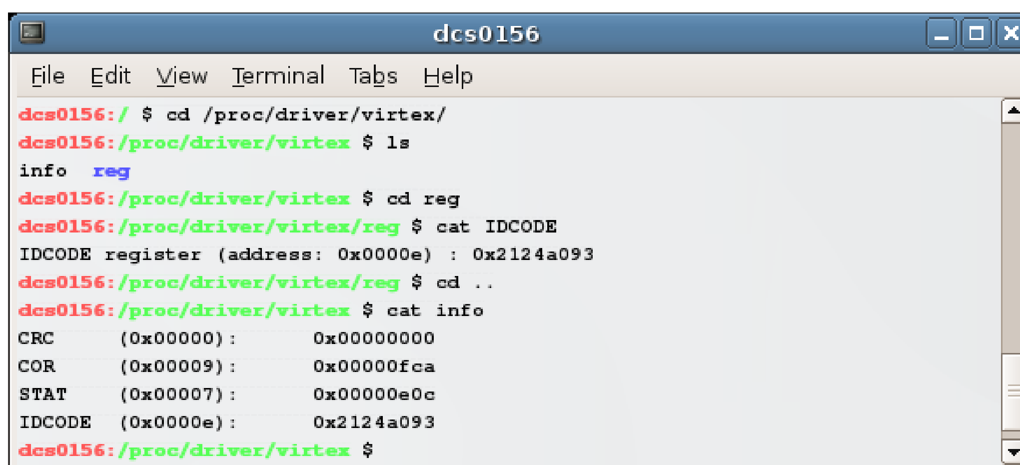
  Der finner man en fil `info` og en mappe `reg.` `info` inneholder noen av registrene med tilhørende verdier. Kommando for å lese denne filen er:

  ```
  cat info
  ```

  Mappen `reg/` inneholder filer for alle registrene. Kommandoen for å lese en fil er:

  ```
  cat <registernavn>
  ```

  Registernavn spesifiserer hvilket register man vil lese.



Skjermdump 2: Bruk av prosessfilsystemet

8.1.3. Skriving til registre

Skriving til registre er foreløpig bare implementert i RCU-shellet. Kommandoen er:

```
sm wr <registeradresse> <data>
```

Registrene er adressert internt på Virtex enheten fra 0 til 14. Ikke alle disse registrene går an å skrive til. Dette er spesifisert i manualen til Virtex enheten. Registeradressen må oppgis i desimaltall, data må oppgis i hexadesimal og må ikke være større enn 32bit/4byte.

8.1.4. Initiell konfigurasjon av Virtex enheten

Driveren har også funksjonalitet for å gjøre initiell konfigurasjon av Virtex enheten. Kommandoen er:

```
cat bitfil.bit > /dev/virtex
```

Filen `bitfil.bit` inneholder data som skal skrives til Virtex enhetens konfigurasjonsminne.

8.2. Drifts- og vedlikeholdsdokumentasjon

Driveren er laget spesifikt for DCS kort; det vil dermed ikke gi noen mening i å kjøre den på noe annet system enn et DCS kort som er koblet til et RCU kort.

Filene som trengs for å installere driveren er ikke noe annet enn selve driver modulen, `virtex_driver.o`. Denne lastes inn i kernelen på vanlig måte; med kommandoen

```
insmod <filsti>/virtex_driver.o
```

For å verifisere at driveren er lastet inn kan man kjøre kommandoen `lsmod`. Hvis man ser seg nødt til å fjerne driveren vil dette typisk bli gjort med kommandoen `rmmod`.

Siden buss-linjene til DCS kortet blir brukt til ulike ting er man nødt til å spesifisere at man skal slå på SelectMAP modus. Kommandoen for dette er i skrivende stund `rcu-sh selectmap enable`.

I vårt prosjekt har vi dannet basisen for driveren som vil bli utviklet videre, ved en senere anledning. Når koden modifiseres er det noen ting man bør være oppmerksom på. F.eks. er det lite feilsjekking av kontrollordet(dvs. hvilke enhetspinner som er satt og ikke), men vi har lagt tilrette for å gjøre det enkelt å implementere dette. Vi har latt være å implementere dette fordi det går på bekostning av effektiviteten til ARM-cpuen, og fordi det er tvilsomt at slik feilsjekking er nødvendig.

`ioctl` magicnumber og driverens majornumber(nummer unikt for hver driver) bør forandres i det ferdige produktet. Begge disse er nå «eksperimentelle» og kan medføre komplikasjoner ved kjøring.

`ioctl` kall for å gjøre det mulig å skrive/lese sekvenser av ord på softwarenivå har blitt lagt til, men det må også legge til en `ioctl` for å «toggle» lese/skrivemodus før denne funksjonaliteten er 100%.

9. OPPSUMMERING, DISKUSJON OG KONKLUSJONER

Da vi begynte på oppgaven hadde vi svært lite kjennskap til programmeringsspråket C, og ingen kjennskap til Linux kernel programmering. Vi fikk heller ikke begynt på oppgaven før to uker etter at vi skulle begynt, siden det var litt frem og tilbake i henhold til hvilken oppgave vi skulle gjøre. Da vi først spurte om hva oppgaven besto av fikk vi beskjed om at det var drivere for Linux. Siden ALICE prosjektet er rimelig stort er det mange forskjellige oppgaver som vi kunne gjort. I de innledende møtene vi hadde med arbeidsgiver fikk vi et par alternativer. Ett av dem var å lage noe om prosessering av logger fra FEE-server, som skal være et kontrollsystem for Front-end Elektronikken. Vi gikk midlertidig for en mer hardware orientert oppgave. Den endelige kravspesifikasjonen på denne mottok vi dagen før vi skulle presentere forprosjektet

Kombinasjonen av en utsatt start og lite kunnskap om hvordan vi skulle gjøre det førte til at vi måtte skrive koden mens vi lærte. Dette førte til at mye av dokumentenasjonen vi skulle hatt klar før vi begynte ikke kunne lages. For eksempel er det vanskelig å skrive om valg av løsningsforslag hvis man ikke har en klar ide om hvordan det kan gjøres.

Vår første iterasjon var å få laget en driver som bare fikk kontakt med Virtex enheten. Vi fikk det til på et vis, men i ettertid viste det seg at vi ikke hadde fått de riktige hardware adressene, slik at vi ikke hadde fått oppnådd noe som helst.

Målet for vår andre iterasjon var å få lest fra et register. På dette tidspunktet hadde vi gjort alt i selve driveren, slik at vi ikke hadde noe software som skulle bruke driveren. Det vil si at alle oppgavene ble utført direkte fra driveren. Det vi skulle ha gjort var å få inn I/O kontroll på et tidlig stadie, men på det tidspunktet hadde vi ikke nok kunnskap til å få det implementert. Dette har også med oppbyggingen i boken vi fulgte, Linux Device Drivers. Der var I/O kontroll ikke et tema før ute i kapittel 5.

Mot slutten av andre iterasjon støtte vi på problemer som virket uforståelige for oss. All kode så ut til å stemme med Virtex-II Pro-manualens prosedyre for å lese et register. Etter en stund med et oscilloskop hvor vi målte hardwarelinjene direkte på kortet fikk vi oppnådd målet vårt, å lese fra et register. Feilen var at vi leste fra data-in linjene og ikke data-ut linjene. Dette var en av flere «små» feil som til tider skapte mye hodebry for oss. Disse feilene var, sammen med at det hele tiden var mye nytt å sette seg inn i, hovedårsaken til at vi i første og andre iterasjon ble liggende et stykke bak fremdriftsplanen. Heldigvis klarte vi hver gang å hente oss inn igjen mot slutten av iterasjonene, slik at fremdriftsplanen ble fulgt.

I tredje iterasjon laget vi en tilleggsmodul til RCU-shellet, slik at man kunne aksessere driveren og lese/skrive til registre. Da dette var gjort, og siden vi hadde i store trekk oppfyllt kravene til oppgaven, bestemte vi oss for å få inn litt ekstra funksjonalitet. Dette inkluderte å få inn IO kontroll slik at vi kunne få kontakt med driveren og utføre operasjoner gjennom software. I tillegg laget vi et proc-filsystem som foreløpig bare kan lese registre, samt et testprogram som kan lese ut en ramme og skrive den til fil. I utgangspunktet ønsket vi å få sistnevnte funksjon inn i RCU-shellet, men dette krevde at dataene ble skrevet ut som char (bokstaver), noe vi ikke fikk tid til å implementere. Testprogrammet gir likevel en god pekepinn på hvordan data fra rammer kan hentes ut fra driveren,

og kan være nyttig å kikke på ved videreutvikling.

Driveren vi har laget inneholder alle de atomære instruksjonene utenom `abort`, som er den eneste begrensingen i forhold til kravspesifikasjonen. Det finnes dog en *busy_handler()* funksjon som har noe av den samme funksjonaliteten. Abort instruksjonen falt bort fordi vi valgte å prioritere å lese fra rammer, og å implementere et proc-filsystem som vi tror kan ha stor nytteverdi, spesielt dersom det implementeres funksjoner for rammer og skriving til registre.

Underveis har vi støtt på en del av de risikoene vi beskrev i risikolisten. Vi måtte endre sannsynlighet til en av risikoene etter at vi opplevde at nøkkelpersoner på instituttet ikke alltid var tilgjengelige for oss. Dette problemet førte til at vi ikke alltid fikk oppklart uklarheter med en gang, men det var aldri noe stort problem siden vi kunne kommunisere over email.

Et større problem var tiden siden vi kom sent igang, men heldigvis klarte vi å komme oss i mål allikevel. Kanskje kunne vi ha kommet enda lengre hvis vi hadde hatt bedre tid.

Det har også vært en del problemer med nettverket på IFT. Det har ikke ført til store problemer for oss, men har vært mer en irritasjon fordi vi har måtte logge inn og ut for å få filene på hjemmeområdet tilgjengelig. En av pc'ene vi har jobbet med også har hatt en del større problemer, plutselig har den avsluttet alle programmene og det har ikke vært mulig å starte dem på nytt igjen. Heldigvis har vi hatt to egne bærbare pc'er som vi har kunnet jobbe med hele tiden. I tillegg mistet vi arbeidsloggen vår vi hadde online, og vi mistenker enten at leverandøren av webhotellet vi brukte slettet den med en feiltakelse eller at noen hacket den. Det ble tatt daglig backup av alle filene på serveren, men desverre ikke av databasen som inneholdt dagboken. Vi håpte at leverandøren hadde backup, men desverre hadde de også bare backup av filene på serveren.

9.1. Avslutning

Det er fortsatt en del som gjenstår i driveren, men vår oppgave var først og fremst å lage noe som det kunne bygges videre på. Det å lære seg å lage en driver og forstå hvordan Virtex fungerer er en prosess som tar tid. Vi føler at det er først nå, når prosjektet nærmer seg slutten, at vi har begynt å få teken på driverprogrammering. Hadde vi hatt sommeren til disposisjon tror vi at vi kunne ha fått utrettet mye. For eksempel kunne vi implementert lese og skriving av rammer i RCU-shellet og proc filsystemet, samt skriving til registre i proc for å nevne noe.

Vi synes prosjektet har vært utfordrende, men samtidig har det vært interessant og morsomt å jobbe med. Til tider har det vært ganske tungt, men alt i alt er vi fornøyde. Vi søkte utfording og hadde lyst å lære noe nytt. Det å jobbe med hardware og Linux drivere var noe vi ikke hadde erfaring med fra før. Forhåpentligvis har vi gjort et arbeid som vil være nyttig for IFT og ALICE.

10. LITTERATURLISTE

Rubini, Alessando et al. Linux Device Drivers, 2nd Edition. Sebastapol, 2001. ISBN 0-59600-008-1.

Bovet, Daniel P. et al. Understanding the Linux Kernel, 2nd Edition. Sebastapol 2002. ISBN 0-596-00213-0.

Free Software Foundation. Gnu «`make`» manual. Sist lest: juni 2006 http://www.gnu.org/software/make/manual/make.html.

Kernighan, Brian W. et al. The C Programming Language, 2nd Edition. New Jersey, 1988. ISBN 0-13-1103790-9.

Krawutschke, Tobias. DCS low-level software. Sist lest: juni 2006. http://frodo.nt.fh-koeln.de/%7Etkrawuts/dcs_documentation.pdf

Love, Tim. ANSI C for Programmers on UNIX Systems. Sist lest: juni 2006. http://www-h.eng.cam.ac.uk/help/documentation/docsource/teaching_C.pdf

Muow, Erik (J.A.K.). Linux Kernel Procfs Guide. Sist lest: juni 2006. http://old.kernelnewbies.org/documents/kdoc/procfs-guide/lkprocfsguide.html

Richter, Matthias et al. The Control System for the Front-End Electronic of the Alice Time Projection Chamber. *Real Time Conference, 2005. 14th IEEE-NPSS*

Salzman, Peter Jay et al. The Linux Kernel Module Programming Guide. Sist lest: juni 2006. http://tldp.org/LDP/lkmpg/2.4/html/lkmpg.html

Xilinx, Inc. Virtex-II Pro and Virtex-II Pro X FPGA User Guide (UG012). Sist lest: juni 2006. http://direct.xilinx.com/bvdocs/userguides/ug012.pdf

Xilinx, Inc. Virtex FPGA Series Configuration and Readback (XAPP138). Sist lest: juni 2006. http://www.xilinx.com/bvdocs/appnotes/xapp138.pdf

11. VEDLEGG

Vedlegg A: Ordliste

Vedlegg B: Fremdriftsplan/Gantt

Vedlegg C: Risikoliste

Vedlegg D: Øvrige Diagrammer

Stikkordregister

A

Active Partial Reconfiguration........................8

ALICE..5

ALICE detektor...5

assert..21

B

bitstrøm...22

buss-linjene på SelectMAP grensesnittet......21

D

deassert...21

debugger...30

desynch...23

Detecor Data Link Source Interface Unit........6

Detector Control System...............................7

driver...19

F

FAC...24

Field-Programmable Gate Array.....................6

Front-End Kort..5

I

I/O kontroll..24

I/O minne...20

I/O port..20

ioctl..24

K

kernel space...25

kernel-log...30

kernelbeskjed...30

Konfigurasjon av Virtex.........................29, 34

kontrollord..21

L

LHC...4

M

major nummer...24

minor nummer...24

N

NOOP..23

O

Oversikt over registre..................................22

P

padframe...25

proc oppføring...26

Proc-filsystemet...25

Pseudokode for les register funksjon............23

Pseudokode for å lese ett 32 bits ord............22

R

RCU-shell..25

Readout Control Unit.....................................5

S

SelectMAP...9

Status Register Readback Sequence.............23

Stråling..7

synkroniseringsord.......................................22

T

Tidsprojeksjonskammeret..............................5

U

user space..25