NTNU
Det skapende universitet

KybOS

Et mikrokjerne- og meldingsbasert
sanntidsoperativsystem

Eirik Wold Solnør

Master i kybernetikk og robotikk
Innlevert: juni 2016
Hovedveileder: Amund Skavhaug, ITK

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

KybOS

Eirik Wold Solnør

Juni 2016

Mastergrad
Institutt for Teknisk Kybernetikk
Norges Teknisk-Naturvitenskapelige Universitet

Veileder: Førsteamanuensis Amund Skavhaug

Forord

Dette er en masteroppgave skrevet våren 2016, ved instituttet for teknisk kybernetikk. Oppgaven var å sette seg inn i teori, planlegge og implementere en prototype av et mikrokjerne- og meldingsbasert operativsystem som kan distribueres over flere noder. Arbeidet som er gjort er en fortsettelse av prosjektoppgaven til forfatteren, [12].

Denne rapporten er rettet mot personer som har god generell kunnskap om programmering, innebygde datasystemersystemer og operativsystemer.

Trondheim, 2015-12-19

Eirik Wold Solnør

Takk til

Takk til Amund Skavhaug for å ha vært en kunnskapsrik og entusiastisk veileder.

Takk til Amund Murstad som utførte responstidtester på KybOS ved av en responstester som han har arbeidet med i sitt prosjekt.

Takk til Simen Sollihøgda og Tom Meland Pedersen for kreative innspill når jeg trengte det og en ekstra hånd under testing.

E.W.S.

Sammendrag

I denne rapporten beskrives KybOS, et operativsystem for en Raspberry Pi 2, samt en del verktøy som har blitt brukt under utviklingen. KybOS er et mikrokjerne og meldingsbasert operativsystem med gode sanntidsegenskaper, som skal kunne videreutvikles til å være distribuert over flere noder. Målet er å oppnå sømløs kommunikasjon mellom prosesser, uavhengig om de skal kommunisere med prosesser som er på samme node, eller en annen. Dette skal være oppgaver som operativsystemet må ta seg av. Grunnen til at det ble valgt å bruke en mikrokjerne-arkitektur er at det er sikrere ved å holde kritiske deler av operativsystemet, kjernen, minst mulig. Ved å bruke meldingsbasert kommunikasjon abstraherer man vekk mange kompliserte sammenhenger som oppstår når prosesser deler minne, og det er også en kommunikasjonsparadigme som egner seg godt når en skal distribuere systemet over flere noder. Det har også blitt gjort tester på responstid og tiden det tar å bytte mellom prosesser.

Kjernen har i stor grad blitt implementert. Kjernen okkuperer en megabyte av minne nederst i det virtulle minnet, mens prosesser kan kjøre i resten av det virtulle adresserommet. Hovedpunkter i den ferdige kjernen kan ses i figur 1. Det er også planlagt at det skal være mulig å videreutvikle dette operativsystemet slik at det kan operere som et distribuert system over flere noder. Dette er ikke i like stor grad implementert, men lagt til rette for. Kjernen er stort sett skrevet i C, men med noe assembler som er nødvendig når en må ha full kontroll over registrene i prosessoren.

Store utfordringer har vært å problemsøke lavenivå kode, hvor det er vanskelig å se hva som skjer å prosessoren og i minnet. Det har også gått mye tid til å utvikle drivere for enheter som er helt nødvendig for videre utvikling. Det har derfor ikke blitt igjen så mye tid til å implementere den distribuerte delen av operativsystemet.

Rapporten slår fast at den utviklede kjernen har i stor grad blitt implementert, men er på ingen måte ferdig. Testene som har blitt utført viser at responstiden ikke er god nok. Den er stort sett god, men noen responser er utliggere og tar svært lang tid. Dette er katastrofalt for sanntidsegenskapene for operativsystemet. KybOS har likevel bedre responstider enn mange andre operativsystemer på en Raspberry Pi 2. En sammenligning med tester utført på UbuntuMate viser at KybOS har langt bedre sanntidsegenskaper.

Dette dokumentet er beregnet til å veiledet fremtidige utviklere for å forstå både hensikten og tankegangen bak KybOS, samt hvordan KybOS fungerer, slik at de kan utvikle systemet videre.

- Kontekstbytter
- Avbrytende multitasking
- Prosesser med eget adresseområdet
- Meldingsbasert kommunikasjon
- Et drivergrensesnitt
- Enkel planlegging for kjøring av prosesser
- Feildeteksjon
- Laste filer fra langtidsminne.

Figur 1: Hovedtrekk av funksjonalitet som er implementert i kybOS

Summary and conclutions

This report describes KybOS, an operating system for the Raspberry Pi 2, as well as tools that have been used during the development. KybOS is a microkernel and message-based real-time operating system, which kan be further developed to be distributed across multiple nodes. The aim is to achieve seamless communication between processes, regardless whether they should communicate with processes on the same node or on another. This should be tasks that the operating system must handle. The reason why it was chosen to use a microkernel architecture is that it is safer by keeping the critical parts of the operating system kernel, to the smallest size possible. By using message-based communication the kernel abstracts away many complex relationships that occur when processes share memory, and it is also a communication paradigm that is very well suited when the aim is to deploy the system across multiple nodes. It has also been done tests on the response time and the time required to switch between processes.

The core has largely been implemented. The kernel occupies one megabyte of memory in the lower virtual memory, while processes run in the rest of the virtual address space. Highlights in the finished core can be seen in figure 2. It is also planned that it should be possible to develop this operating system so that it can operate as a distributed system across multiple nodes. This has not been implemented, but facilitated. The kernel is mostly written in C, but with some assembler that is required when the kernel must have complete control over the registers of the processor.

Major challenges has been to debug low level code, where it is difficult to see what happens in the processor and memory. There has also been a lot of time used to develop drivers for devices that are absolutely necessary for further development, for instance the SD driver. It has therefore not been much time to implement the distributed part of the operating system.

The report states that the developed core has largely been implemented, but is by no means finished. The tests that have been conducted have shown that the response time is not good enough. It is mostly good, but some responses are outliers and takes a very long time. This is disastrous for the real-time characteristics of the operating system. Even so, KybOS has better response times than many other operating systems on a Raspberry Pi 2. A comparison with tests carried out on UbuntuMate shows that KybOS have far better real-time properties.

This document is intended to be a guide for future developers to understand the purpose and reasoning behind KybOS and how KybOS works, so they can further develop the system.

- Context switching.
- Preemptive multitasking.
- Process with their own address space.
- Message based communication.
- A driver interface.
- A simple scheduler for the processes.
- Error detection.
- Ability to load from the SD-card.

Figur 2: Highlights of the functionality that is implemented in KybOS

Innhold

1 Innledning

Hensikten med denne oppgaven har vært å lage et mikrokjerne- og meldingsbasert operativsystem med sanntidsegenskaper som skal kunne distribueres over flere noder. Arbeidet har i stor grad bestått av å implementerer og lære om hardware ved å lese dokumentasjon samt prøve og feile-metoden. Det har også gått med tid til å lese litteratur om operativsystemer og dere grunnleggende mekanismer

1.1 Bakgrunn

Det er blitt viktig å koble sammen flere noder til et nettverk for å samle informasjon og sette pådrag på enheter. Kommunikasjonen over disse nodene blir gjort i applikasjons-laget. Dette kan være vanskelig og komplisert, slik at utvikling blir lang og dyr. Det ville vært svært mye enklere om en kunne ha koblet disse nodene sammen ved å kjøre dem over et operativsystem som tok seg av all kommunikasjon. Samtidig er det viktig å ha sanntidsegenskaper som gjør det mulig å si noe vettugt om hvordan systemet vil oppføre seg med hensyn på responstid. Det finnes mange operativsystem som er bygget rundt meldingsbasert kommunikasjon og en mikrokjerne arkitektur, men ingen har særlig god støtte for å distribuere systemet. Dette blir selvfølgelig mer og mer viktig i en verden hvor ”tingenenes internett” blir utbredt.

Det finnes mange operativsystemer som er meldingsbaserte og bygget rundt en mikrokjerne-arkitektur. Operativsystemer som QNX og GNU Mach kan nevnes som eksempler, men begge har ikke god nok støtte for å distribuere systemet over flere noder. Dessuten kan GNU Mach bare kjøre på en x86 arkitektur. Det finnes mye god litteratur om operativsystemer. For å gjøre seg kjent med grunnleggende mekanismer til operativsystemet er [13]. Spesielt de tidlige kapitlene om tråder og prosesser har vært til stor hjelp for å forstå mekanismene for flerprosesskjøring.

1.2 Oppgaven

Oppgaven er å lage kjernen til kybOS, et operativsystem som er mikrokjerne- og meldingsbasert. Oppgaven består av å sette seg inn i operativsystemers oppgaver og arkitektur, hvordan man kan bruke hardware til å implementere funksjonalitet, og å implementere kjernen i størst mulig grad. Det burde også planlegges en arkitektur for resten av operativsystemet. Det må lages slik at det er mulig å bygge et slikt operativsystem rundt kjernen. Først må en finne en passende platform, før en kan begynne å å implementere kjernen. Kjernen må ha egenskapene beskrevet i figur 1.1.

- Avbrytende multitasking
- Prosesser med egne adresseområder
- Meldingsbasert kommunikasjon mellom prosesser
- Drivergrensesnitt
- Sanntidsegenskaper
- Mulighet for å laste inn nye prosesser

Figur 1.1: Egenskaper ved KybOS

Til slutt er det ønskelig å teste sanntidsegenskapene systemet. To egenskaper som er ønskelig å teste er hvor lang tid KybOS bruker på å reagere på ekstern stimuli, og hvor lang tid det tar å bytte prosess. Her er det ønskelig å få en god distribusjon over responstidene slik at man kan gi noen garantier om responstiden til operativsystemet.

1.3 Tilnærming

Det er store mengder litteratur om operativsystemer. Man kan muligens si at det derfor er best å planlegge hele systemet først, og deretter implementere det etterpå. Det viser seg at dette raskt blir umulig når systemer blir av en stor nok kompleksitet, samtidig som at det er umulig å absorbere all kunnskapen i litteraturen før en starter å implementere. I tillegg er det svært lange og detaljerte dokumentblader som beskriver utviklingsplatformen og eksterne enheter på opptill flere tusen sider hver. Det er nødvendig å gjøre seg kjent i denne dokumentasjonen og lære seg hvordan man gjør enkle, men svært viktige operasjoner.

Det viser seg at den beste måten å arbeide på er å tilegne seg teoretisk kunnskap og å implementere parallelt. Slik får man brukt teorien fortest mulig, samt at man blir kjent i hardware. Dette var en god måte å arbeide på da man kunne splitte tiden mellom teori og praksis.

Arbeidsprosessen kan deles inn i faser:

1. Bestemme seg for platform og få den startet opp slik at man kan begynne å implementere systemet.
2. Klare å bruke avbrudd, prosessormodus, bytte kontekst mellom tråder og få multitasket mellom dem
3. Lage funksjonalitet for å sende meldinger mellom disse trådene.
4. Bruke en Memory Management Unit(MMU) til å lage et kart av minnet. Det er ikke mulig å bruke MMU-en til å gi trådene sitt eget adresserom enda fordi de er kompilert sammen med kjernen.
5. Utvikle en SD-driver som gjør det mulig å laste filer fra SD-kortet og opp til minnet.
6. Lage funksjonalitet for å lese .elf filer og starte prosesser av dem. Her brukes MMU-en til å lage egne adresserom for prosessene.
7. Implementere et drivergrensesnitt.
8. Teste sanntidsegenskapene til systemet.

1.4 Strukturen av rapporten

Rapporten inneholder 15 kapitler. Disse omhandler følgende:

1 Innledning Her gis det en bakgrunn for problemer og det forklares hva som skal gjøres i resten av rapporten.

2 Teori Her introduseres noen teoretiske begrep som brukes i resten av rapporten.

3 Platform Her forklares det hvordan platformen ble valgt og hvilke verktøy som ble brukt for at det skulle være mulig å utvikle for denne platformen.

4 Starte opp systemet Her forklares det hvordan KybOS initialiserer hardware, setter opp C-runtime, og starter viktige moduler.

5 Grensesnitt mot PC og feilsøking Dette er kapittelet som viser hvordan man kan koble opp utviklingsplatformen mot en PC slik at man kan kommunisere med systemet. Her vises også hvordan man kan feilsøke ved hjelp av JTAG.

6 Memory Management Unit Forklares hvordan KybOS bruke en Memory management Unit. Det forklares hvordan oversettingstabellen lages og hvilke register som må konfigureres for å skru på MMU-en.

7 Minnet Beskriver hvordan KybOS holder styr på minnet og fordeler det til prosesser.

8 Prosesser og multitasking Viser hvordan KybOS implementerer og legger til rette for prosesser og hvordan kjernen multitasker mellom dem. Dette kapittelet tar for seg all kode som håndterer avbrudd.

9 Kommunikasjon mellom prosesser Dette er kapittelet hvor det forklares hvordan man kan implementere meldingsbasert kommunikasjon med systemkallene send og receive.

10 Drivere Her forklares hvordan grensesnittet til drivere fungerer.

11 SD kort driver Å laste opp filer fra SD kortet er essensielt for å starte nye prosesser ved operativsystemets oppstart. Derfor inkluderes en SD- og FAT-driver i kjernen.

12 Postboksen Postboksen er en funksjonalitet for å kunne kommunisere med andre prosessorer og GPU-en.

13 Arkitektur Her beskrives sekvenser av meldinger som må implementeres for at operativsystemet skal kunne distribueres over flere noder.

14 Tester Det er gjort to tester på KybOS som tester sanntidsegenskapene til operativsystemet. Utførelsen og resultatene av disse presenteres her.

15 Diskusjon I dette siste kapittelet så oppsummeres hva som er gjort i løpet av prosjektet, og hva som er oppnådd. De viktige valgene som ble tatt i dette prosjektet diskuteres, og det blir foreslått hvordan man kan jobbe videre med kjernen.

2 Teori

Her dekkes flere tema som er nødvendig å forstå for å bygge et meldingsbasert mikrokjerne-operativsystem. Målet er å bygge et operativsystem der all kommunikasjon gjøres ved hjelp av meldinger. Det skal ikke være forskjell, fra en prosess sitt synspunkt, på å sende en melding til sin egen node som en annen node. Det skal være mulig å ha flere planleggings-algoritmer. Et viktig aspekter at prosessene må være avbruddbar noe som gir kortest mulig responstid. Videre lesning i disse tema kan finnes i [13] og [14]. Dette kapittelet er i stor grad hentet fra [12].

2.1 Kjernen

Operativsystemet kan deles inn to deler. Kjernen, som er kjernen av operativsystemet og tar seg av en liten mengde essensielle oppgaver, og alt annet. Kjernen kjører i kjernemodus. Det vil si at kjernen har tilgang til å endre viktige kontrollregistre i prosessoren som styrer oppførselen til systemet. Kjernen tilbyr systemkall til den andre delen av operativsystemet. Resten av systemet kjører i brukermodus, hvor man ikke har tilgang til slike viktige registre.

En mikrokjernearkitektur betyr at mest mulig skal kunne hånteres i prosesser utenfor kjernen. Kjernen gjør bare noen få elementære oppgaver. Viktige oppgaver til kjernen vil være å håndtere avbrudd, legge til rette for prosesser, planlegge hvilke prosesser som skal kjøre, kommunikasjon mellom prosessene, og håndtere minnet. Kjernen vil ikke være avbrytbar, det vil si at det er svært viktig at alt kjernen gjør ikke er langvarige oppgaver. Ved å holde kjernens oppgaver kortest mulig forbedres responstiden til hele systemet. Oppgaver som å håndtere eksterne enheter, implementere protokoller og lignende blir lagt til drivere som kjører utenfor kjernen.

2.2 prosesser

En nødvendig del av et operativsystem er å lage prosesser. Prosesser er en abstraksjon som operativsystemet lager. Prosesser er en instans av et program som har sitt eget adresserom, og blir eksekvert uavhengig av andre prosesser. Det er viktig å implementere en form for beskyttelse mellom prosesser, slik at en prosess ikke kan skrive eller lese i en annen prosess sitt minneområdet. En prosess skal ikke kunne merke at det blir hurtig byttet inn og ut av prosessoren: Det kan bare merke at den har blitt byttet inn og ut ved å se at tid har passert. Et viktig aspekt med et distribuert system er at en prosess må kunne ”pakkes ned” og sendes til en annen node. Dette er for å for eksempel distribuere arbeidsmengden over flere noder.

2.2.1 Drivere

Et spesialtilfelle av prosesser er drivere. Disse er prosesser som skal håndtere enheter som er koblet til systemet. For å gjøre dette må drivere ha tilgang til input-output adresser som ligger i minnet, og de må få vite når et avbrudd som har sin kilde i enheten skjer. Drivere må også være tilgjengelige for andre prosesser med gjenkjennelige navn. En god måte å løse dette på er at når en driver registrerer seg vil operativsystemet sende meldinger til driveren hver gang driverens enhet genererer et avbrudd.

2.2.2 Process Control Block

En prosess vil ha en del viktig informasjon om seg som lagres i kjernen i en Process Control Block. Noen elementer som ofte er inkludert i en PCB er vist i 2.1. Disse er ofte implementert som en dobbeltlenket liste i heapen til kjernen, i følge [13].

Identitet En unik ID som er assosiert med prosessen. Dette kan bestå av et nummer, eller kan være noe mer komplisert. En god løsning på dette i et distribuert system er å la IP-adressen være en del av identiteten.

tilstand Beskrivelse av hvilken tilstand prosessen er i. Tilstander som trengs i et meldingsbasert operativssytem er: Klar/Kjører, Venter-på-melding og, om man bruker synkronisert kommunikasjon, venter-på-å-sende.

Prioritet Angir hvor høy prioritet prosessen har. Høyere prioritet gjør at prosessen vil bli høyere prioritert av planleggeren. Nøyaktig hvilken policy planleggeren bruker er implementasjonsavhengig og kan variere.

Kontekst data Når operativsystemet bytter kontekst må man lagre alle registrene i prosessoren til minnet.

Informasjon som brukes av planleggeren Avhengig av hva slags planlegger policy man har, vil planleggeren være nødt til å lagre noe informasjon

Informasjon om virtuelt minne Operativsystemet må vite hvor i det virtuelle adresserommet prosessen kjører.

Informasjon om fysisk minne Operativsystemet må vite hvor i det fysiske minnet prosessen kjører. Dette inkluderer selve programkoden og minnet, hvor stor heapen er og størrelsen på stakken.

Figur 2.1: Process Control Block

2.3 Bytte kontekst

Å bytte mellom prosesser kalles å bytte kontekst. Konteksten til en prosess er alle generelle registre i prosessoren, stakk peker, link registeret, prosessor modus og mer basert på hvordan operativsystemet er bygget. Det er ofte også nødvendig å ta vare på innstillinger i MMU-en (Memory Management Unit). For å bytte kontekst er det nødvendig å lagre konteksten til en prosess i en PCB, sette prosesstilstand til Blokkert, kjøre planleggeren for å finne en prosess som kan byttes inn, sette tilstanden til den nye prosessen til kjører, og til slutt laste inn den nye konteksten.

2.3.1 Planleggeren

Planleggeren har ansvar for hvilken prosess som kjører. Det finnes mange algoritmer for hvordan dette kan gjøres. Planleggeren må holde styr på hvor mye hver enkelt prosess har kjørt og velge ut en egnet prosess som skal kjøre. Den må velge prosesser som er i tilstand Klar, og unngå å velge prosesser i tilstand Blokkert. En enkel algoritme er å alltid kjøre den prosessen som har høyest prioritet.

1. En prosess gjør et system kall og oppgir et minneområdet hvor en melding ligger.
2. Kjernen kopierer minneområdet til kjernens minneområdet, og innkapsler det i noe metadata, som for eksempel hvilken prosess den ble sent fra, størrelse og flagg.
3. En annen prosess gjør et systemkall for å få meldingen. Prosessen oppgir et minneområde den vil ha meldingen kopiert til.
4. Kjernen kopierer minnet over til motagende prosess.

Figur 2.2: Hendelsesforløp for å sende en melding

2.4 Meldingsbasert kommunikasjon og synkronisering

Det er viktig å implementere kommunikasjon mellom prosessene med meldinger, og ikke med delt minne. Det er viktig at planleggeren kopierer minnet, og ikke bare en minnelokasjon. Dette er for å hindre at en prosess får en peker som går inn i minneområdet til en annen prosess. Dette vil, ikke bare være en katastrofe for sikkerhetshensyn, men MMU-en vil også gi generere et avbrudd fordi en prosess prøver å laste i fra minne utenfor sitt området. For å sende en melding må hendelsesforløpet beskrevet i 2.2 skje.

Operativsystemet må også kunne sjekke om meldingen er på denne noden, eller en annen node, og deretter sende meldingen til riktig node.

2.4.1 Pålitelig eller Upålitelig

Viktige aspekter er om meldingene skal være pålitelig eller upålitelig. Pålitelig kommunikasjon vil så at operativsystemet garanterer at meldingene kommer fram, om mulig.

Pålitelig kommunikasjon kan være bra fordi det gjør det mye lettere å utvikle applikasjoner, men det er også svært komplisert. Det er enkelt nok når det er bare én node, men om meldingen skal sendes over internettet er det ikke mulig å garantere om meldingen kommer fram eller ikke.

Om en ønsker å ha en liten mikrokernelbasert OS kan det være lurere å bruke upålitelig kommunikasjon, og heller la utviklere legge til den funksjonaliteten i applikasjonen om de trenger det.

2.4.2 Synkronisert eller asynkron

Ved asynkron kommunikasjon vil en prosess gå til tilstand Klar så snart operativsystemet har kopiert meldingen over til sitt minneområdet. Alternativt kan man bruke synkronisert kommunikasjon, da vil sendende prosess gå til Klar først når operativsystemet har kopiert meldingen over til mottagende prosess. Det kan være vanskelig å bruke synkronisert kommunikasjon over flere noder, spesielt om kommunikasjonen er upålitelig.

Send Sende en melding til en prosess.
Send driver Sende en melding til en navngitt driver.
Receive Motta en en melding.
Spawn Lage en prosess. Allokere minne og lage PCB.
Yield Prosessen gir avkall på nåværende tidssegment og lar en annen prosess overta.
Kill Avslutte en gitt prosess. Frigi minnet, og slette PCB.
Exit Avslutte denne prosessen. Avslutte prosessen kallet er gjort fra.
Mmap Prosesser får tilgang til den fysiske adressen, slik at prosessen kan skrive til bestemte minneadresser.
Driver register Registrer en prosess som driver. Prosessen vil få meldinger ved bestemte avbrudd.
SRBK Kalles når en prosess må øke størrelsen på heapen. Denne er viktig for at kjernen skal vite hvor i det fysiske minnet en prosess kjører og bruker minne til variabler.

Figur 2.3: Systemkall som er essensielle

2.5 Systemkall

brukerprosesser kan kommunisere med operativsystemet ved hjelp av systemkall. Det er et mål om å holde systemkall så få som mulig, for å få en minst mulig kjerne. Systemkallene som er nødvendige i kybOS er beskrevet i figur 2.3.

Systemkall er gjort ved å generere et programmvareunntak.

1. En prosess som kjører i brukermodus kaller en funksjon som er tilbudt av et bibliotek.
2. Biblioteksfunksjonen genererer et unntak, med informasjon om hva slags systemkall det er, og noen andre argument avhengig av hva som trengs, liggende i gitte registre i prosessoren.
3. Unntaks håndtereren finner ut hvilket kall det er ved å se på registrene, og håndterer oppgaven
4. Avhengig av hvilket systemkall vil kjernen enten gå tilbake til den kallende prosessen, eller gjøre bytte kontekst til en annen prosess.

3 Platform

Før en kan begynne å lage et KybOS må en finne en passende platform å implementere. Det er også nødvendig å finne et sett med utviklingsverktøy som er gode å jobbe med og kompatible med platformen. Dette kapittelet omhandler valgent av utviklingsplatform og hva slags innmat den har.

Valget falt på en Raspberry Pi 2, med utviklingsverktøy arm-none-eabi fra GNU. Ved å bruke et C-biblotek Newlib, spares mye tid på å slippe å lage en egen implementasjon av en heap og annen nyttig funksjonalitet. Dette kapittelet er i stor grad hentet fra [12].

3.1 Valg av platform

Før en kan begynne å implementere kjernen må en bestemme seg for hvilken platform det er hensiktsmessig å utvikle for. Alternativer kan være en x86-arkitektur, eller et ”system on a board”, for eksempel en Raspberry Pi eller Beaglebone Black. Viktige trekk er beskrevet i 3.1:

- Støtte for bruker-/kjernemodus i prosessor.
- 2 eller flere tilgangsprivilegier.
- bytte mellom modusene.
- En Memory Management Unit(MMU) for å muliggjøre separate minneområdet for prosesser, og beskyttelse.
- kostnad.
- Hvor vanskelig det er å utvikle.
- Prosessorkraft og størrelse av minnet.
- online community.
- Muligheter for å feilsøke.

Figur 3.1: Viktige trekk ved platform

Basert på dette ble det valgt en Raspberry Pi 2b. Den er bygget rundt en bcm2836 mikrokontroller fra Broadcom, som har 4 ARM cortex-a7 prosessorer. A-serien har kapabilitet for å utføre kompliserte oppgaver, som for eksempel å kjøre et operativsystem. Den har flere prosessormodi deriblant bruker- og kjernemodus. Hver av kjernene har en MMU som gjør det mulig å implementere separate minneområder for hver prosess. Den er også lett tilgjengelig og er svært utbredt slik at å finne hjelp og eksempelkode på internettet ikke bør være så vanskelig.

Raspberri Pi 2 har også mer enn nok kraft. Standard systemklokkefrekvens er 700 MHz, men den kan konfigureres opp til 1 GHz. Den har også 1 GB minne, noe som er mer enn nok for rammene av dette

Figur 3.2: Raspberry pi 2

Avbrudd	Verdi	Modus
Reset	0x0	Supervisor
Undefined	0x4	Undefined
Software Interrupt	0x8	Supervisor
Prefetch abort	0x0c	Abort
Data abort	0x10	Abort
Interrupt	0x18	IRQ
Fast interrupt	0x1c	FIQ

Figur 3.3: Predefinert lokasjon av forskjellige avbrudd

prosjektet. Det er også mulig å feilsøke ved hjelp av et JTAG-grensesnitt noe som gjør feilsøking mye enklere. Det har har også muligheten til å bruke en av de mange GPIO pinnene til å gi output til omverdenen.

3.2 Prosessoren

Et viktig element av et operativsystem er å kunne implementere brukermodus og kjernemodus. I prosessoren, cortex-a7, er det implementer 2 prioritetsnivå. I PL0 er det ikke mulig å skrive til viktige registre som styrer systemet. Dette er brukermodus. PL1 er kjernemodus, og kan skrive til stort sett alle registre. Hvilket prioritetsnivå systemet er i er definert av hvilken prosessormodus. Man kan lese i [7] om hvordan prosessoren fungerer.

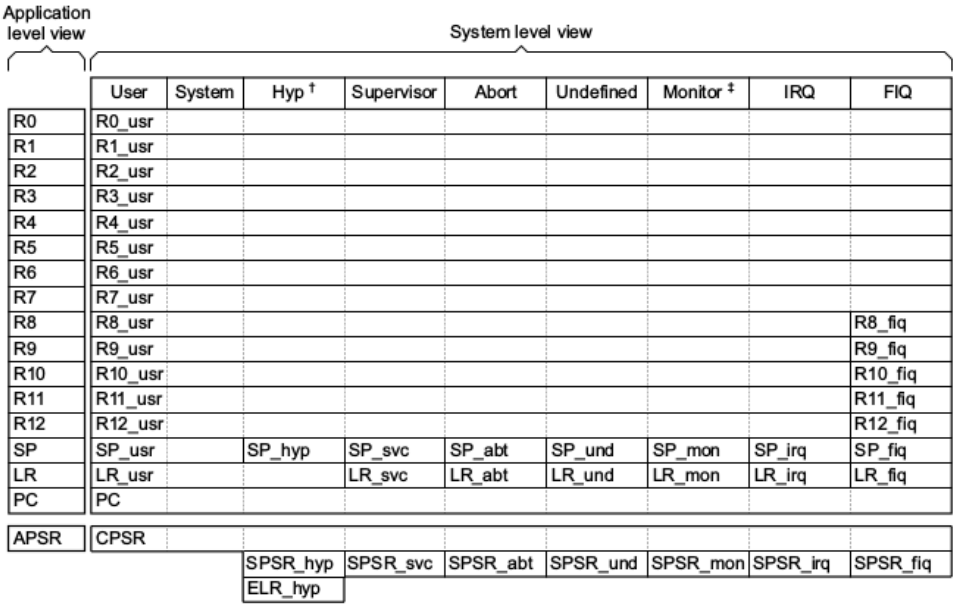
Prosessormodus	Prioritetsnivå	CPSR bit 0-4
User (usr)	PL0	0b10000
Fast Interrupt (FIQ)	PL1	0b10001
Interrupt (IRQ)	PL1	0b10010
Supervisor (SVC)	PL1	0b10011
Monitor (mon)	PL1	0b10110
Abort (abt)	PL1	0b10111
Undefined (und)	PL1	0b11011
System (sys)	PL1	0b11111

Tabell 3.1: Prosessormodi

3.2.1 Prosessormodus

Det er 9 prosessormodi på cortex-a7. Disse er listet i tabell 3.1. Hvilken modus prosessoren er i er styrt av registeret Current Program Status Register(CPSR) bit 4 til 0 (CPSR.M). Å bytte mellom modi er ganske enkelt om man er i PL1, der er bare å skrive til CPSR.M med den verdien du ønsker å bytte til. Om man er i PL0 er det ikke mulig å skrive til CPSR. Måten man bytter modus da er å generere et avbrudd. Ved et avbrudd går prosessoren til relevant modus. se figur 3.3.

Det er en skyggeregistre for hver modus. Dette kan ses i figur 3.4. Her ser man at de fleste modi har skyggeregistre for stakk pekeren (SP), link registeret (LR), og Saved Program Status Register (SPSR).



Figur 3.4: Skyggeregistre for arm cortex-a7. Dette er figur B1-2 fra [7]

3.2.2 Avbrudd

Bcm2836 har en avbruddskontroller som styrer avbrudd. Det er 7 typer avbrudd: reset, undefined, software interrupt, prefetch abort, data abort, interrupt og fast interrupt. Når prosessoren oppdager en avbruddskilde setter den programpekeren til en predefinert verdi, som kan sees i figur 3.3. I denne plassen av minnet kan man putte en instruksjon som gjør et hopp til en funksjon som hånterer avbruddet.

Idet et avbrudd skjer dette: Prosessoren bytter modus ut ifra hvilket avbrudd som skjedde, altså vil skyggeregistrene til hver modus bli brukt. Programpekeren vil bli lastet inn i link registeret og CPSR vil også bli lastet inn i SPSR. Programpekeren vil så bli satt til den predefinerte verdien.

3.2.3 System modus

System modusen er noe spesiell fordi den har ingen skyggeregistre og er i PL1. Dette er fordi at det skal være mulig å få tak i brukermodus' stakk peker og linkregister i de andre modusene. Om man hadde vært nødt til å gå til User mode, så ville en kommet ned i PL0, altså uten mulighet for å komme tilbake til PL1.

3.3 Memory Management Unit

En svært viktig del av platformen er Memory Management Unit(MMU). Den oversetter blokker av virtuelt minne til fysisk minne slik at det er mulig at hver prosess får kjøre i sitt eget adresserom, samtidig som det styrer hvem som har lov til å skrive hvor i det virtulle minnet. Dette er svært nyttig, da det kan brukes til å beskytte kjernens minneområde og beskytte prosessene mot hverander ved å bare gi prosessene tillatelse til å skrive til sitt eget minne.

Raspberry Pi 2s MMU har funksjonalitet for flere blokkstørrelser, se figur 3.5. i KybOS ble det bare brukt 1 Section-segmenter for å forenkle MMU-operasjoner. Dette gjør at kjernen og alle prosesser

Supersection Beskriver 16MB blokker av minnet.
Section Beskriver 1MB av minnet.
Large Page Beskriver 64KB av minnet.
Small page Beskriver 4KB av minnet

Figur 3.5: Minnesegmenter

får et segment på 1 MB hver, dette er mer enn nok med god margin. Som beskrevet i figur 3.3 vil avbrudd føre til at programpekeren blir satt til en verdi nær null. Dette gjør at kjernen må okkupere den nederste section-segmentet i det virtuelle minnet. I resten av det 4GB store virtuelle minnet kan prosesser kjøre.

3.4 Verktøy

Det første en må gjøre er å få tak i en kryss-kompilator og binutils som lager kompatibel kode mot cortex-a7 prosessorene. Disse er platformavhengig, og vi kan derfor ikke bruke standardkompilatoren på en PC til å kompilere for en Raspberry Pi. Det er nødvendig å bruke en krysskompilator som kompilerer kode til en annen platform enn den kjører på. Det er også lurt å kunne sette opp systemet slik at man kan bruke en versjon av standardbibiloteket til C. I dette prosjektet er Newlib brukt for til dette formålet.

3.4.1 GNU Binutils

GNU binutils er gruppe av binære verktøy som er svært nyttige for å softwareutvikling.

ld GNU linkeren.
as GNU assembleren.
addr2line Konverterer adresser til filnavn of linjenummer.
dlltool Verktøy for å bygge og bruke DLL-er (Dynamic-link library).
nm Lister opp symbolene i en objekt fil.
objcopy Kopierer og oversetter objektfiler.
objdump Viser informasjon om objektfiler

Figur 3.6: viktige GNU binutils som blir brukt i dette prosjektet

3.4.2 GNU Compiler Collection

GNU Compiler Collection(GCC) er en kompilator som kompilerer kode ned til maskinkode. GCC bruker mange av verktøyene i binutils for å bearbeide koden, og må linkes mot den under kompilering.

3.4.3 Newlib

Man ønsket også å kunne bruke standardbibloteket til C. Man kan kompilere og linket et bibliotek som heter Newlib. Dette er et bibliotek som er implementert mest mulig uavhengig av hardware, slik at det kan brukes uten et operativsystem. Det er noen funksjoner som er avhengig av noen grunnleggende systemkall i bunnen, men disse kan man implementere selv, eller gjør slik at de returnere en feilverdi når de kalles. Dette er veldig nyttige, fordi man får tilgang på funksjoner som malloc, free, osv. Dokumentasjonen til Newlib finnes her [4]

3.4.4 Kompilerer en krysskompilator

Det enkleste å bruke en GCC versjon kalt arm-none-eabi. det første leddet er ”arm” fordi raspberry pi 2 har en arm arkitektur, bruker en ARM cortex-a7. Det neste leddet er hvilket OS den skal kompilere for. Systemet skal ikke kjøre over noe operativsystem, så det er ”none”. Det siste leddet står for ”Embedded Application Binary Interface”. Først så må man kompilere en binutils. Deretter kan man kompilere gcc som må linkes mot binutils. Pakker som må lastes ned eller installeres er beskerevet i 3.7. Det finnes gode bruksanvisninger på hvordan man kan kompilere egne krysskompilatorer blant annet her [2].

Gnu Bison en parser generator.
Flex Et verktøy for å generere programmer som gjør mønstergjenkjenning.
GNU GMP Et bibliotek for aritmetikk med vilkårlig presisjon for heltall, rasjonelle tall og flytall.
GNU MPFR Er et C bibliotek for flyttallsoperasjoner med vilkårlig presisjon, basert på GNU GMP.
GNU MPC Et C bibliotek for komplekse tall med vilkårlig presisjon. bygget på GNU MPFR.
Texinfo Er en typesetting syntax. Brukes for å generere informasjon.
ISL Et C bibliotek for å manipulere set og heltall. Bygget på GNU GMP
CLooG Som brukes for å optimaliser kode.

Figur 3.7: Pakker for å kompilere egen GCC for krysskompilering

For å kompilere binutils, kopier mappene for ISL og ClooG inn Binutils nedlastingen og bruk GNU make for å kompiler binutils. Her må du oppi hvilken platform du kompilerer for, og generelle flagg som du ønsker å bruke. Deretter kan du kompilere GCC: Flytt GMP, MPFT, MPC, ISL og ClooG inn i GCC nedlastingen og bruk make for å kompilere. Her må du også oppgi hvilken platform vi ønsker å utvikle mot. Om alt gikk riktig vil binærkoden som kompilatoren produserer være kompatibel med raspberry pi-en.

Alle versjoner er ikke nødvendigvis kompatible. Forfatteren forsøkte først å bruke den nyeste versjonen av GCC(5.2.0). Da den ikke fungerte rullet ble 4.8.4 forsøkt. Denne versjonen fungerte fint. Det er uvisst hva som gjør at 5.2.0 ikke fungerte, men etter flere forsøkt ble det slått fast at det enkleste videre var å bruke 4.8.4.

4 Starte opp systemet

I dette kapittelet beskrives oppstartsekvensen til Raspberry Pi 2, hvordan man kan kompilere kode for denne platformen, og til slutt beskrives hvordan kjernen gjør tidlig initialisering og gjør det mulig å gå fra å skrive i assembler til skrive kode i C, som er et mye høyere nåvå språk.

I dette kapittelet beskrives hvordan systemet starter opp, og hvordan man kan kompilere kode for Rapberry Pi 2 ved hjelp av GNUs arm-none-eabi utviklingsverktøy. Dette kapittelet er i stor grad hentet fra [12].

4.1 Oppstartsekvens

Før man kan begynne å skrive i C kode må man klare å starte opp systemet. Raspberry Pi 2 har et mini sd-kort som systemet starter fra. Sd-kortet må være FAT formatert og inneholde kode som brukes til oppstarten. I Raspberry Pi-en er det en CPU og en GPU. Det er GPU-en som utfører oppstarten, for å så gi kontrollen videre til prosessoren.

1. Først laster den og kjører en fil som heter bootcode.bin. Dette er en program som lastes inn i L2 hurtigbufferen og skrur på SDRAM minnet i systemet, og deretter laster inn neste steg.

2. start.elf har som hovedoppgave å laste en fil som er kalt kernel.img inn i SDRAM adresse 0x8000. Den gjør også noe initialisering, som å sette frekvensen til systemklokka og videoresolusjon (om man skal bruke det). Det er mulig å konfigurere systemet med en fil config.txt. Etter dette steget gir GPU-en kontrollen videre til CPU-en ved å sett programmpekeren til 0x8000.

Det er ikke mulig å skrive bootcode.bin, loader.bin og start.elf selv, fordi man må kjøpe lisens for å få tilgang til dokumentasjonen til GPU-en. Det man derimot kan gjøre, er å laste ned oppstartsfiler fra raspberry-pi sin github side, [3]. Her kan man laste ned de nødvendige filene. Filene man trenger å laste ned er bootcode.bin og start.elf. Legger man disse filene over på sd-kortet vil *"act"* LED-et blinke under oppstarten. Ved å kompilere et eget program ned til en fil kalt kernel.img og sette det på sd-kortet sammen med oppstartsfilene vil programmet bli lastet opp på Pi-en ved adresse 0x8000.

4.2 Kompilere egen kode

Om vi bare prøver å kompilere egen kode uten videre vil vi få en feil under kompileringen. Om man kompilerer en enkel fil:

```
int main(void){
```

```
    return 0;
}
```

med kommandolinjekall:

```
arm-none-eabi-gcc -Wall -o kybOS kybOS.c
```

Vil man få en feilmelding fra linkeren:

```
/tmp/ccIL64M2.o: In function `main':
kybOS.c:(.text+0x0): multiple definition of `main'
/tmp/ccEtjHJM.o:kybOS.c:(.text+0x0): first defined here
../arm-none-eabi/lib/libc.a(lib_a-exit.o): In function `exit':
../newlib/libc/stdlib/exit.c:70: undefined reference to `_exit'
collect2: error: ld returned 1 exit status
```

Dette betyr at linkeren ikke finner symbolet _exit, som kalles ifra funksjonen exit. Exit er en bibliotekfunksjon som kalles når main kaller return. Exit igjen kaller _exit, som er et kall til operativsystemet. Exit er i dette tilfellet fra biblioteket Newlib. Det er et bibliotek som prøver å implementere mest mulig uten å bruke operativsystemet eller være hardware avhengig, men i dette tilfellet kaller exit _exit. Fordi systemet ikke har noe operativsystem er naturligvis ikke _exit definert heller. Exit vil i dette tilfelle ikke ha noen funksjon, så det enkleste løsningen er å erstatte exit med en ny exit:

```
void exit(void){
    while(1){};
}
```

Når man kompilerer dette vil ikke kompilatoren klage, og filen kompilerer. Et problem slik det er blitt gjort til nå, er at kompilatoren antar at det er oppstartsfiler som gjør en del initialisering for at C runtime fungerer. For å komme rundt det og få kompilert noe kode som fungerer, kan man legge til flagget -nostartfiles.

Om man kompilerer med

```
arm-none-eabi-gcc -Wall -o kybOS -nostartfiles kybOS.c
```

vil man få en advarsel:

```
/usr/lib/gcc/arm-none-eabi/4.8.2/../../../arm-none-eabi/bin/ld: warning:
 cannot find entry symbol _start; defaulting to 0000000000008000
```

Det vil fungere helt fint fordi 8000 er plassen vi vil ha koden vår til. Problemet nå er at filen som lages av kompilatoren er i elf formatet. Elf er et filformat som linux bruker for å innkapsle maskin kode i meta-data. Dette brukes til diverse formål av operativsystemet, men fordi Raspberry Pi-en er uten et operativsystem kan dette fjernes. For å ta ut bare den rene maskinkoden kan man bruke objcopy på elf filen, og legge det i en fil kernel.img:

```
arm-none-eabi-objcopy kybOS.elf -O binary kernel.img
```

Filstørrelsen kan ses ved å bruke ls -l. Her vil kernel.img være litt mindre enn kybOS.elf, fordi vi har fjernet all metadata:

```
$ ls -l
-rwxr-xr-x 1 eirikws eirikws 33658 Nov  3 12:02 kybOS.elf
-rwxr-xr-x 1 eirikws eirikws    48 Nov  3 12:05 kernel.img
```

Her ser man selve binærkoden for programmet tar 48 bytes, mens .elf fila tar 33 kb. Størrelsen for kernel.img ser riktig ut, men kybOS.elf ser alt for stor ut. Dette er på grunn av en feil i standard linkerskriptet som brukes når man ikke gir linkeren et linkerskript som argument.

4.2.1 Linkerskript

Den beste måten å se hvorfor filen er så stor er å bruke et verktøy kalt arm-none-eabi-nm. Om man først kompilerer kernel.elf, og så bruker nm til å hente ut symboltabellen til en ny fil kybOS.nm:

```
arm-none-eabi-nm ./kybOS.elf > ./kybOS.nm
```

```
1  00010020 T __bss_end__
2  00010020 T _bss_end__
3  00010020 T __bss_start
4  00010020 T __bss_start__
5  00010020 T __data_start
6  00010020 T _edata
7  00010020 T _end
8  00010020 T __end__
9  0000800c T exit
10 00008000 T main
11 00080000 N _stack
12          U _start
```

Her ser man at main er linket til minneadresse 0x8000. Men det som er veldig rart er at .data delen av linkerskriptet er linket til adresse 10020. Dette må være en feil i linkerskriptet, det er ingen grunn til å et stort gap mellom .data og .word. Koden vår består av to funksjoner; Exit og main. Exit starter på adresse 0x800c og er den siste funksjonen i minnet. Det neste symbolet i minnet er .data delen (som foreløbig er tom) ved 0x10020. Ved å observere at $0x10020 - 0x800c = 8014$ ser man at det er et justering på ca. 0x8000. Hvis en tar en titt på standard linkerskrip ved å gi arm-none-eabi-gcc flaget -wl,-verbose ser vi at det er en linje 106 som ikke gir mening:

```
. = ALIGN(CONSTANT (MAXPAGESIZE)) + (. & (CONSTANT (MAXPAGESIZE) - 1));
```

Scriptet vil prøve å oppjustere til neste page, uten noen grunn. Om man antar at dette er en feil og kopierer outputet fra terminalen inn i min egen linker fil, kommenterer bort denne linjen og gir linkerscriptet som input, vil ting fortsatt fungere, og filstørrelsen vil gå ned til noen få hundre.

Viktige ting å legge merke til i linker skriptet er:

```
ENTRY(_start)
```

Som sier er starten av programmet er symbolet _start, her starter altså programmet. En annen viktig linje er:

```
PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x8000));
    . = SEGMENT_START("text-segment", 0x8000);
```

Som sier at starten at fila skal legges til adresse 0x8000. Linkeren definerer også et område kalt .bss. Dette området er variabler som skal initialiseres til null. Dette er noe som vi må gjøre selv.

4.3 Oppstarten av KybOS

Linkerskript-et definerte _start som inngangspunkt i programmet. Da kan vi enkelt lage et funkson kalt _start i assembler og starte derfra. Hovedpunker er:

1. Kopierer instruksjoner som gir et hopp til avbrudssbehandlere til riktig adresse. Disse instruksjonene ligge på linje 38-45, og kopieres til adresse null og oppover ved linje 51-56.
2. Plassere ut stakker for hver eneste prosessormodi som brukes. Disse må være innenfor den første 1 MB i minnet. Dette skjer ved linje 59-84.
3. Skrive .bss området til null. Dette gjøres i en C funksjon kalt _cstartup.

```
1  .equ    CPSR_MODE_FILTER,       0x1F
2  .equ    CPSR_MODE_USER,         0x10
3  .equ    CPSR_MODE_FIQ,          0x11
4  .equ    CPSR_MODE_IRQ,          0x12
5  .equ    CPSR_MODE_SVR,          0x13
6  .equ    CPSR_MODE_ABORT,        0x17
7  .equ    CPSR_MODE_UNDEFINED,    0x1B
8  .equ    CPSR_MODE_SYSTEM,       0x1F
9  .equ    CPSR_IRQ_INHIBIT,       0x80
10 .equ    CPSR_FIQ_INHIBIT,       0x40
11 .equ    CPSR_THUMB,             0x20
12 .equ    STACK_SVR,              (0xf0000) // these adresses are within
13 .equ    STACK_UNDEF,            (0x100000)// the first 1 MB block of memory
14 .equ    STACK_IRQ,              (0xf8000) // because of 1 MB page table!
15 .equ    STACK_ABORT,            (0xfc000)
16 .equ    STACK_INIT,             (0x100000)
17 .equ    STACK_SYSTEM,           (0xfa000)
18
19 /*
20         Put the vector table in the ram at 0x8000.
21         At startup the PC will start at 0x8000, which is the first
22         instrucion of _start.
23 */
24 _start:
25     ldr pc, _reset_h
```

```
26      ldr pc, _undefined_instruction_vector_h
27      ldr pc, _software_interrupt_vector_h
28      ldr pc, _prefetch_abort_vector_h
29      ldr pc, _data_abort_vector_h
30      ldr pc, _unused_handler_h
31      ldr pc, _interrupt_vector_h
32      ldr pc, _fast_interrupt_vector_h
33
34  /*
35      Create a table of constants with an entry for each vector. The compiler
36      will keep the labels relative to PC
37  */
38  _reset_h:                           .word   _reset_
39  _undefined_instruction_vector_h:    .word   undefined_instruction_vector
40  _software_interrupt_vector_h:       .word   software_interrupt_vector
41  _prefetch_abort_vector_h:           .word   prefetch_abort_vector
42  _data_abort_vector_h:               .word   data_abort_vector
43  _unused_handler_h:                  .word   _reset_
44  _interrupt_vector_h:                .word   interrupt_vector
45  _fast_interrupt_vector_h:           .word   fast_interrupt_vector
46
47  _reset_:
48      // We enter execution in supervisor mode!
49
50      // Copy _start and the table from 0x8000 to 0x0000
51      ldr     r0, =_start
52      mov     r1, #0x0000
53      ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
54      stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
55      ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
56      stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
57
58      //  setup the stack pointer for the interrupt mode
59      mov r0, #(CPSR_MODE_IRQ | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
60      msr cpsr_c, r0
61      mov sp, #(STACK_IRQ)
62
63      // setup Undefined
64      mov r0, #(CPSR_MODE_UNDEFINED | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
65      msr cpsr_c, r0
66      mov sp, #(STACK_UNDEF)
67
68      //setup Abort
69      mov r0, #(CPSR_MODE_ABORT | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
70      msr cpsr_c, r0
71      mov sp, #(STACK_ABORT)
72
73      //  setup the user/system stack pointer. These modes share registers.
74      //  switch to system mode because the we are able to switch back!
```

```
75      mov r0, #(CPSR_MODE_SYSTEM | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
76      msr cpsr_c, r0
77      mov sp, #(STACK_SYSTEM)
78      //  switch back to supervisor mode and set the stack pointer
79      //  to init stack. The stack will work down, and the heap up.
80      mov r0, #(CPSR_MODE_SVR | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
81      msr cpsr_c, r0
82
83      //  set the stack pointer at some point in RAM.
84      mov sp, #(STACK_INIT)
85
86      // now that the stacks are set up, it is possible to call C funksjons
87      //  Setup the c runtime
88      bl      _cstartup
89
90      // Should never come here, but if it does, loop forever for debug purpuses
91  _inf_loop:
92      b       _inf_loop
```

```
1   extern int __bss_start__;
2   extern int __bss_end__;
3
4   extern void kernel_main( unsigned int r0,
5           unsigned int r1, unsigned int atags );
6
7   void _cstartup( unsigned int r0, unsigned int r1, unsigned int r2 )
8   {
9       int* bss = &__bss_start__;
10      int* bss_end = &__bss_end__;
11      //    Initialize the _bss section to zero
12      while( bss < bss_end )
13          *bss++ = 0;
14
15      kernel_main( r0, r1, r2 );
16
17      //  Trap
18      while(1)
19      {
20          /* EMPTY! */
21      }
22  }
```

_cstartup itererer seg gjennom .bss området og setter det til null. __bss _start __og __bss _end __er symboler definert i linkerskriptet. Når det er gjort er C runtime startet og man kan kalle en funksjon kernel_main som er starten av kjernen. Det er nå mulig å skrive kode i C og forvente at koden gjør som den skal, da alle variabler er initialisert til riktige verdier.

Når C-runtime er startet kan KybOS starte opp og initsialisere flere ting i en kernel_main funksjon.

Denne initsialiserer flere moduler, før den laster inn noen prosesser.

```
1  void kernel_main( unsigned int r0, unsigned int r1, unsigned int atags ){
2      _enable_interrupts();
3      uart_init();
4      uart_puts("kernel start!\r\n");
5      drivers_init();
6      jtag_enable();
7      arm_timer_init();
8      mmu_init_table();
9      mmu_configure();
10     cpu_control_config();
11     // start fat filesystem on the SD card
12     fs_init();
13
14     // loading processes
15     process_load("prog1.elf", 20, CPSR_MODE_USER, (process_id_t){1});
16     process_load("prog2.elf", 20, CPSR_MODE_USER, (process_id_t){2});
17
18     //  starting them
19     process_start( (process_id_t){2});
20     process_start( (process_id_t){1});
21
22     set_preemptive_timer(1);
23     _SYSTEM_CALL(YIELD,0,0,0);
24 }
```

I linje tre til tolv initialiserer funksjonen moduler som KybOS trenger for å kjøre. Uart_init gjør det mulig å skrive tekst gjennom en UART forbindelse til en annen datamasking. Jtag_enable er en annen, mer kraftig måte å feilsøke på. Drivers_init gjør det mulig for prosesser å registrere seg som drivere. Mmu_init_table og mmu_configue klargjør MMU-en for å skrus på. Cpu_control_config skrur på MMU-en, hurtigbufferen og en del andre innstillinger som styrer hvordan prosessoren oppfører seg. Fs_init initialiserer et FAT filsystem som gjør det mulig å laste filer fra SD kortet.

Det neste som skjer er at funksjonen laster opp noen programmer fra SD-kortet, og starter dem i bruermodus og prioritet lik 20. Til slutt skrur funksjonen på avbrytende multitasking, slik at timer-avbrudd fører til et contekst bytte mellom prosesser. Deretter tvinger den frem et kontekst bytte ved å kalle systemkallet YIELD. KybOS er da startet opp. Prosesser kjører, mens kjernen kjører i svært korte øyeblikk når et avbrudd finner sted.

5 Grensesnitt mot PC og feilsøking

For å kunne drive effektiv utvikling er det nødvendig å kunne feilsøke systemet. Det er to måter å feilsøke som har blitt bruke under utviklingen. Den første er å printe ut tekst gjennom en UART forbindelse med en datamaskin. Dette er metoden som har blitt brukt mest da den er enkel og rask å bruke. Den andre måten å feilsøke på er å bruk et JTAG grensesnitt. JTAG er et mye mer kraftig feilsøkingsverktøy som gir kontrollen over prosessoren til en ekstern kontroller. Slik kan man gå instruksjonvis gjennom koden, få innblikk i register verdier og mer. Dette er en mye tregere måte å feilsøke på da det kreves mer utstyr og og tid. Likevel er JTAG svært nyttig for å finne stygge feil i lavnivå kode. Deler av dette kapittelet kommer fra [12]

5.1 UART

UART(Universial asynchronous receiver/transmitter) er en enkel bus over to datalinjer. Raspberry pi-en har en mini UART-kontroller og en PL011 UART. I dette prosjektet PL011 brukt. For å få UART til å fungere må man først koble opp noe som kan konvertere UART til et grensesnitt mot PC-en, og deretter slå på UART i programvaren. Dokumentasjonen kan finnes i dokumentasjonen til BCM2835, [10].

5.1.1 Hardware

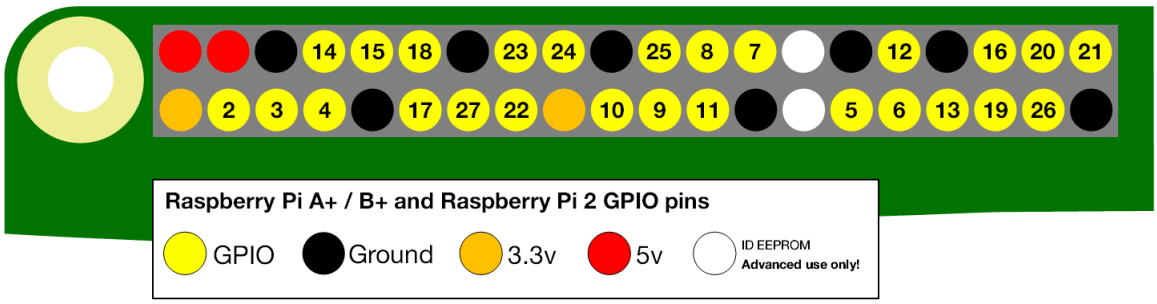
UART er plassert i minnet ved 0x3f20100, og er koblet til GPIO pinne 14 og 15, hvor 14 er TXD og 15 er RXD. Disse kablene kan kobles opp mot en uart-til-USB kabel, for eksempel en TTL kabel slik som i figure 5.1. Her er den røde kabelen koblet mot 5 volt, svart mot jord, hvit mot TXD og grønn mot RXD. Denne kabel driver systemet uten noen annen strømforsyning.

5.1.2 Programvare

For å skru på UART er det noen få ting som må gjøres. Her ønsker man å ha 115200 i baud rate. For å få til dette må man kalkulere divider og fraction , og skrive det til IBRD og FRDB. Etter dette må en gjøre noe konfigurering. PL011 har en FIFO buffer på 8 symboler på input og output. Den har muligheten til å generere et avbrudd basert på hvor full den er.

Her er UART konfigurert slik at den genererer et avbrudd når input-bufferen er 1/8 full, altså hver gang UART mottar et symbol. Output bufferen er ikke konfigurert for å lage avbrudd. Det er altså avbruddsdrevet å motta.

Figur 5.1: TTL kabel



Figur 5.2: GPIO pins

```
1  void uart_init( void ){
2      uart_get()->CR = 0x00000000; //      turn everything off for starters
3      // Control signal to disable pull up/down
4      // and delay for 150 clock cycles
5      get_gpio()->GPPUD = PULL_UPDOWN_DISABLE;
6      delay(150); // disable pull up/down for uart pins and delay
7      get_gpio()->GPPUDCLK0 = UART_PINS;
8      delay(150); // remove control signal and clocking
9      get_gpio()->GPPUD = 0;
10     get_gpio()->GPPUDCLK0 = 0; //  clear impending interrupts
11     uart_get()->ICR = 0x7ff;
12     //  set the baud rate:
13     //  divider = uart_clock/(16 * baud)
14     //  fraction = ( fraction_part * 64 ) + 0.5
15     //  UART_CLOCK = 3000000, baud = 115200
16     //  Divider = 3000000 / (16 * 115200) = 1.627 ~ 1
17     //  fraction = (0.627 * 64) + 0.5 = 40.6 ~ 40
18     uart_get()->IBRD = 1;
19     uart_get()->FBRD = 40;
20      // configure UART:
21     uart_get()->LCRH = WORD_LEN_8BIT | FIFO_DISABLE;
22     // set receive interrupt fifo level to 1/8 FIFO level
23     uart_get()->IFLS |= RECEIVE_IRQ_FIFO_18;
24     uart_get()->IMSC   |= RECEIVE_MASK_BIT; //  mask interrupts
25     GetIrqController()->Enable_IRQs_2 |= UART_IRQ;
26     uart_get()->CR = UART_ENABLE | TRANSMIT_ENABLE | RECEIVE_ENABLE;
27 }
```

For å skrive til UART og sende symboler kan man lage seg en funksjon. For å skrive et symbol til output-bufferen må man først sjekke om bufferen er full. Om den er full må man vente til den ikke lenger er full. Dette kan enten gjøres ved avbrudd eller polling. Her er en enkel funksjon implementert men polling. Større funksjoner for å skrive setninger og variabler kan bygges over denne.

```
1  void uart_putc(unsigned char byte){
2      while( uart_get()->FR & TRANSMIT_FIFO_FULL ){}
3      uart_get()->DR = byte;
4      return;
5  }
```

Her er en funksjon som leser ett symbol ved hjelp av polling. Det venter til det er noe i input bufferen, leser symbolet og returnerer det.

```
1  unsigned char uart_getc(){
2      while ( uart_get()->FR & RECEIVE_FIFO_EMPTY){ }
3      return uart_get()->DR;
4  }
```

5.1.3 Koble til fra PC-en

Om man kobler TTL-kabelen til PC-en kan man bruke minicom fra terminalen til å snakke med UART. Fordi systemet ble satt opp til å ha baud-rate 115200, brukes det. For å vite hvilen port man skal bruke kan man bruke dmesg for å se de forskjellige tilkoblingene:

```
$ dmesg | grep usb
[274491.011938] usb 3-1: new full-speed USB device number 2 using xhci_hcd
[274491.140695] usb 3-1: New USB device found, idVendor=067b, idProduct=2303
[274491.140700] usb 3-1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[274491.140703] usb 3-1: Product: USB-Serial Controller
[274491.140705] usb 3-1: Manufacturer: Prolific Technology Inc.
[274491.141835] usb 3-1: pl2303 converter now attached to ttyUSB0
```

Her ser en klart at TTL-kabelen er koblet til /dev/ttyUSB0. Da kan man koble seg til ved hjelp av minicom:

```
$ minicom -b 115200 -D /dev/ttyUSB0
```

5.2 JTAG

En måte å feisøke er å bruke UART til å skrive relevant informasjon, men noen ganger trenger man mye kraftigere verktøy for å feilsøke. Man har lyst til å koble opp GNU gdb mot raspberry pi-en for å sette breakpoints, gå gjennom kode linje for linje, se hva som skjer i forskjellige registere osv. Det er ikke mulig med en enkel UART.

Joint Test Action Group (JTAG) er et slikt grensesnitt for å feilsøke. JTAG kan gjøre alt dette, samt lese/skrive i minnet. Det er derfor et kraftig verktøy. JTAG protokollen beskriver 4 linjer mellom JTAG-adapteren og platformen beskrevet i figur 5.3.

TCK Test clock.
TDI Test data in.
TDO Test data out.
TMS Test mode select

Figur 5.3: JTAG bussen

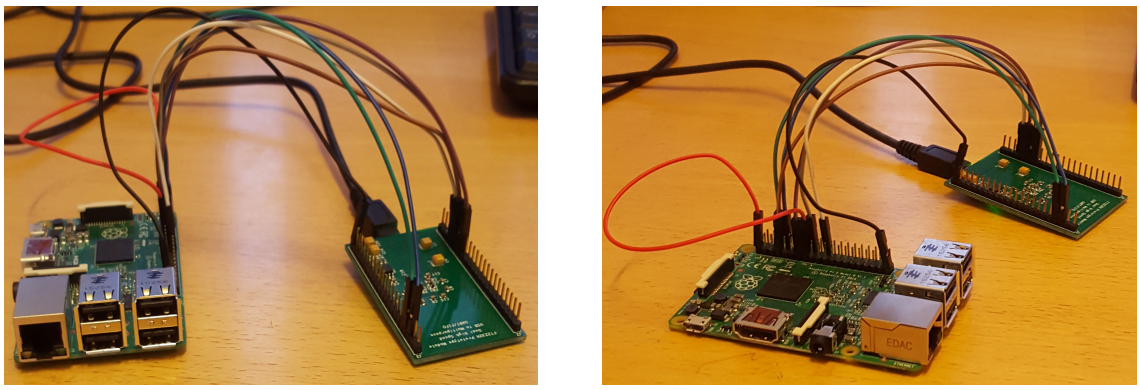
Det første en må gjøre er å skaffe til veie en kontroller som er JTAG- kompatibel med prosessoren til raspberyr pi, cortex-a7. En populær brikke er Future Technology Devices International Ltd. (FTDI) sin FT2232H eller FT4232H. Disse er Brikker med USB drivere for kommunikasjon med en PC, og kan brukes for flere kommunikasjonsprotokoller mot platformen. De er kompatible med mange prosessorer, inkludert De ARM prosessorer. Under dette prosjektet er det brukt en brikke med FT2232H, kalt FT2232H Breakout Module. Den ble kjøp på ebay her `http://www.ebay.com/itm/FT2232H-Breakout-Module-/111714216200`. Den kan også brukes til UART parallelt, slik at TTL-kabelen ikke lengre benyttes.

Koble opp JTAG adapteren

For å koble opp adapteren mot Raspberry Pi-en må man først lese dokumentasjonen til FT2232H for å se hvilke output pin som må kobles, og deretter se på diagrammet til Breakout Module A.1 for å se hvor signalene er ført. Deretter må man se hvor disse skal kobles til på Raspberry Pi-en Dette er beskrevet i figur 5.4. I tillegg må man koble jord over. Da er det lett å se at vi må koble Breakout Module J2 19 med Raspberry Pi pin 22, J2 18 med 7 osv. I tilleg må 3,3 volt (pin nr 1) kobles til GPIO22 (pin 15). Når dette er ferdig koblet vil det se ut slik som i figur 5.5. Raspberry Pi-en bør nå drives av strøm fra usb-porten. JTAG-adapteren drives også en USB port, og kommuniserer gjennom den til en PC.

	FT2232H pin	Breakout Module Pin	Raspberry Pi pin	GPIO nr
TCK	16	J2 19	22	25
TDI	17	J2 18	7	4
TDO	18	J2 17	18	24
TMS	19	J2 16	13	27
TXD	38	J3 3	10	15
RXD	39	J3 4	8	14

Figur 5.4: Signalene må kobles slik. De



Figur 5.5: Raspberry pi koblet opp mot JTAG adapteren.

JTAG programvare for Raspberry Pi 2

På Raspberry pi-en er ikke JTAG bussen eksponert ved oppstart. Man må konfigurere GPIO pinnene til å være i JTAG modus. Dette kan gjøres slik. Det som skjer her er at GPIO pinnene settes riktig modus (JTAG) ved i skrive til GPFSEL0 og GPFSEL2. Dokumentasjon om dette finnes i BCM2835 ARM Peripherals som kan lastes ned fra raspberry pi sin hjemmesider.

```
1 void jtag_enable(void){
2     // set gpio4 to alt 5
3     gpio_t *gpio = get_gpio();
4     gpio->GPFSEL0 &= ~(7 << 12);     // zero out gpio4
5     gpio->GPFSEL0 |= 2<< 12;     //gpio4 to alt5 = ARM_TDI
6     // set other gpios
7     gpio->GPFSEL2 &= ~(
```

```
8                           (7<<6)      //zero gpio22
9                          |(7<<12)     // gpio24
10                         |(7<<15)     // gpio25
11                         |(7<<21));   // gpio27
12
13      gpio->GPFSEL2 |=     (3 << 6)   // alt4 ARM TRST
14                         |(3 << 12)   // alt4 ARM_TDO
15                         |(3 << 15)   // alt4 ARM_TCK
16                         |(3 << 21);   // alt4 ARM_TMS
17      return;
18  }
```

5.2.1 OpenOCD

Målet er å klare å debugge raspberry pi-en med GNU GDB. FOr å gjøre dette må man klare å koble sammen GDB med JTAG adapteren. Dette kan gjøres av OpenOCD som er er et opensource prosjekt som tilbyr et grensesnitt mellom gdb og telnet, og en JTAG-adapter. OpenOCD er en server som oversetter JTAG grensesnittet for GDB, slik at man kan feilsøke raspberry pi-en en PC hvor man kjører GDB og OpenOCD. Dokumentasjonen til OpenOCD kan finnes her [1]

Det OpenOCD gjør når den starter opp er å først lese to konfigurasjonsskript, som sier noe om hvilken adapter den skal brukes mot, og noe om platformen som skal feilsøkes. Deretter verifiserer den at lenken fungerer. Til slutt venter den på at GDB og Telnet skal koble seg opp som klienter.

OpenOCD skript

Det er to skript som trengs: Et for JTAG-adapteren og en for platformen som skal feilsøkes. Adapter skriptet trenger relativt lite informasjon. man må si noe om hva slags grensesnitt adapteren bruker, Device ID, Vendor ID og noe initialisering. Dette er et skript som setter opp OpenOCD for FT2232H. Grensenittet er FTDI (som produserer brikken). Vendor ID, device ID og initialisering kan finnes fra databladet til FT2232H.

```
# The interface is a FTDI device
interface ftdi
# The vendor ID and device ID
ftdi_vid_pid 0x0403 0x6010
# Initial values of FTDI GPIO data and direction registers
ftdi_layout_init 0x0018 0x05fb
ftdi_layout_signal nSRST -data 0x0020
```

Skriptet for Raspberry pi-en er noe mer komplisert. Først setter den opp OpenOCD porter som telenet og gdb kan koble seg på. Deretter setter den opp navn på brettet. Deretter må man sette op 1 DAP (Test Access Port) for hver av de fire prosessorene i brikken. Det er disse DAP-ene som kan feilsøkes når man bruker GDB. DAP-ene må konfigureres, blant annet med hva slags type prosessor det er (cortex-a), gis ID, og muliggjør multiprosessor feilsøking. Til slutt så konfigureres her av DAP-ene slik at når GDB kobler seg mot dem, så utfører JTAG-en noe initialisering slik at GDB-en kan begynne.

```
. #which ports are to be used for gdb and telnet. these are standard ports
telnet_port 4444
gdb_port 3333
adapter_khz 1000
#set chipname
if { [info exists CHIPNAME] } {
        set  _CHIPNAME $CHIPNAME
} else {
        set  _CHIPNAME bcm2836
}
if { [info exists DAP_TAPID] } {
        set _DAP_TAPID $DAP_TAPID
} else {
        set _DAP_TAPID 0x4ba00477
}
# declare a Test Access Port
#  -irlen the number of bits in instruction register
jtag newtap $_CHIPNAME dap -expected-id $_DAP_TAPID -irlen 4 -ircapture 0x01 -irmask 0x0f
# set a target for each of the cpu
set _TARGETNAME0 $_CHIPNAME.cpu0
set _TARGETNAME1 $_CHIPNAME.cpu1
set _TARGETNAME2 $_CHIPNAME.cpu2
set _TARGETNAME3 $_CHIPNAME.cpu3
# ceate a target for each of the cpu
target create $_TARGETNAME0 cortex_a -chain-position $_CHIPNAME.dap -coreid 0 -dbgbase 0x80010000
target create $_TARGETNAME1 cortex_a -chain-position $_CHIPNAME.dap -coreid 1 -dbgbase 0x80012000
target create $_TARGETNAME2 cortex_a -chain-position $_CHIPNAME.dap -coreid 2 -dbgbase 0x80014000
target create $_TARGETNAME3 cortex_a -chain-position $_CHIPNAME.dap -coreid 3 -dbgbase 0x80016000
target smp $_TARGETNAME0 $_TARGETNAME1 $_TARGETNAME2 $_TARGETNAME3
# When a gdb connects, do the cortex_a-spesific instructio dbginit
# dbginit: enable core debugging.
$_TARGETNAME0 configure -event gdb-attach {
```

```
33          cortex_a dbginit
34  }
35  $_TARGETNAME1 configure -event gdb-attach {
36          cortex_a dbginit
37  }
38  $_TARGETNAME2 configure -event gdb-attach {
39          cortex_a dbginit
40  }
41  $_TARGETNAME3 configure -event gdb-attach {
42          cortex_a dbginit
43  }
```

5.2.2 Bruk av GDB, Telned og OpenOCD til å feilsøke

For å koble OpenOCD mot JTAG-adapteren må GPIO allerede være konfigurert til å oppføre seg som JTAG-bussen. Ofte er det vanlig å bruke JTAG til å laste opp kode, men det er ikke mulig fordi JTAG ikke er på ved oppstart. Det som må skje er at først må raspberry pi-en starte opp og konfigurere GPIO. Deretter kan OpenOCD starte. Man kan starte OpenOCD ved dette kommandokalled.

```
$ sudo openocd -f breakout_module.cfg -f bcm2836.cfg
```

Etter dette kan man koble seg opp med telnet.

```
$ telnet localhost 4444
```

Man kan koble gdb mot OpenOCD. Kallet må være gjort fra arm-none-eabi-gdb, som er kompatibel med cortex-a.

```
(gdb)  target extended-remote localhost:3333
```

6 Memory Management Unit

For å behandle minnet er Memory Management Unit (MMU) svært viktig. Den kan brukes til å gi hver prosess et eget adresseområdet slik at prosesser ikke kan skrive eller lese fra andre prosesser minneområdet. Ved å oversette virtuelt minne til fysisk minne kan prosesser tro at de er ved et minneområdet, men egentlig kjøre i et helt annet sted i minnet. Dette er nyttig, for eksempel kan man kompilere et program til å kjøre ett sted i det virtuelle minnet, men være lagret et helt annet sted i det fysiske minnet. Når dette programmet blir lastet opp kan kjernen bruke MMU-en til å gi prosessen et virtuelt minnerom der det starter i null, mens dens fysiske minneadresse er noe helt annet. Dette er veldig nyttig for et operativsystem, både med tanke på å kunne dynamisk laste opp programmer i minnet og kjøre dem, og sikkerhet mellom prosesser og kjernen.

Dette kapittelet beskriver de elementære funksjonene som KybOS. Hvordan man initialiserer MMU-en og hvordan man skriver nye verdier til den. Informasjon om MMU-en kan finnes i [7], spesielt i kapittel B3 Virtual Memory System Structure.

I dette kapittelet blir det introdusert 4 funksjoner:

mmu_init_table som initialiserer oversettingstabellen.

mmu_configure Som konfigurerer registre som tilhører MMU-en.

cpu_control_config som konfigurerer systemkontroll registeret til prosessoren til å skru på MMU-en, hurtigbufferen og mer.

mmu_remap_section som skriver en ny virtuell til fysisk oversettelse til oversettingstabellen, og oppdaterer TLB.

6.1 Oversettingstabellen

For å skru på MMU-en må man først bygge en oversettingstabell. Denne tabellen inneholder informasjon om hvert segment av det virtuelle minnet. Nøyaktig hvilken informasjon som er

Supersection Beskriver 16MB blokker av minnet.
Section Beskriver 1MB av minnet.
Large Page Beskriver 64KB av minnet.
Small page Beskriver 4KB av minnet

Figur 6.1: Minnesegmenter

tilgjengelig i oversettingstabellen er avhengig av hvor store minnesegmenter som brukes. Jo mindre minneområdet er, jo flere bit må brukes til å beskreive den fysiske adressen. Det vil med andre ord bli mindre bit igjen for å beskrive attributter av minneområdet. Derfor krever large page og small page at det er implementert et nivå 2 oversettingstabell som gir nok bit for å beskrive minneområdene. Derfor er kanskje Section typen den enkleste å implementere. Denne typen er implementert i KybOS.

6.1.1 Minnetyper

Strongly Ordered Alle aksesser til minnet må være ferdig før man kan begynne på en ny. Aksesser kan ikke bufre og kan ikke bruke hurtigbufferen.

Device Brukes om områder hvor aksess til minnet kan gi bivirkninger. Aksesser kan bufre men ikke bruke hurtigbufferen.

Normal Kan deles opp i flere grupper basert på hurtigbuffer innstillinger. Kan bufre og bruke hurtigbufferen.

Figur 6.2: Minnetyper

Write-through Write er gjort synkront til både hurtigbufferen og minnet.

Write-back Write er gjort bare til hurtigbufferen til å begynne med. Å skrive til minnet gjøres først når hurtigbuffer-blokken blir flyttet ut av hurtigbufferen.

Write Allocate Ved en write-bom vil hurtigbufferen laste opp minnelokasjonen til hurtigbufferen, fulgt av en write som som denne gangen vil treffe.

No write allocate Ved en write-bom vil hurtigbufferen skrive rett til minnet uten å gå laste opp til hurtigbufferen.

Figur 6.3: Hurtigbufferinstillinger

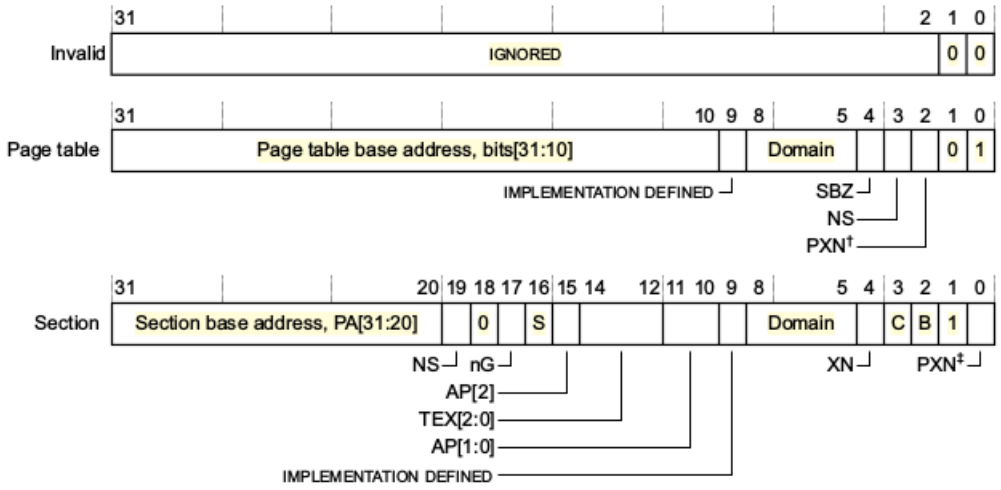
En et minneområde kan kan en av flere minnetypene beskrevet i 6.2. Normal minne kan igjen deles opp i flere typer avhengig av hvordan man bruker hurtigbufferen. figuer 6.3 beskriver hvordan en hurtigbufferen kan fungere. Write-through og Write-back beskriver når en hurtigbuffer skriver til minnet, og Write allocate og no write allocate beskriver når hurtigbufferen leser fra minnet. Write-through er en enkel algoritme, og man kan også være sikker på at all informasjon i minnet er oppdatert. Med andre ord er det sikkert. Write-back, på den andre siden er en hurtigere men mer komplisert algoritme. Den er heller ikke sikkert, fordi hurtigbufferen og minnet ikke nødvendigvis trenger å være konsistent. De to mest vanlige kombinasjonene er write-through og no-write allocate(fordi senere write til samme hurtigbuffer blokker vil fremdeles gå gjennom minnet), og write-back og write allocate(i håpet om at senerere write til samme hurtigbuffer blokk vil bli fanget av hurtigbufferen). Hvilken minnetype er bestemt av bitene TEX, C og B.

6.1.2 Aksesstillatelse

Hvilke prioritetsnivå som har lov til å aksessere, og om hvert prioritetsnivå har bare love til å skrive, lese eller begge.

6.1.3 Domain

Hver minneregion tilhører et Domain. Hver prosess må forholde seg til et domain, som en Manager eller Client. En Client må forholde seg til aksesstillatelsene beskrevet i oversettingstabellen, mens en Manager kan ignorere dem.



Figur 6.4: Bit felt i oversettingstabellen for page og section. figur B3-4 fra [7]

PXN Om prosessoren kan eksekvere kode i området ved PL1.
Bit[1:0] Bestemmer typen: 0b00 Invalid. 0b01 page table. 0b10 section eller supersection.
TEX[2:0], C, B Bestemmer minneregions attributter.
XN Hvis denne biten er 1, kan ikke kode som er i dette område bli eksekvert.
Domain Angir hvilken Domain dette segmentet er en del av.
AP[2], AP[1:0] Aksess tillatelse bit.
S Avgjør om det adresserte minnet er kan deles mellom prosesser.
nG Ikke global bit.
NS Non-Secure bit.
PA[31:20] Hvilken fysiske adresse segmentet peker til.

Figur 6.5: Beskrivelse av bitfelt for section

6.1.4 Delbar

Et minneområdet kan være enten delbar eller ikkedelbar over flere prosessorer. Dette styres av bit S i oversettingstabellen. Dette sikrer at alle data aksesser blant alle prosesser er koherent.

6.1.5 Fysisk adresse

Feltet PA bestemmer hvilken fysisk adresse som den virtuelle minnetsegmentet skal peke til.

6.2 Å lage en oversettingstabell

Den enkleste er å bruke section. En section må være 16MB-aligned i minnet. Section ser ut som i figur 6.4, hvor hvert av feltene er beskrevet i figur 6.5. KybOS starter med en oversettingstabell som gir kjernen tilgang til til det nederste 1 MB av minnet. Denne section-segmentet er bare aksesserbart i kjernemodus. Kjernen må også ha tilgang til innput-output registre som ligger ved adressene 0x3f000000 til 0xffffffff. Disse adressene blir markert som Device-minne og er også bare aksesserbart fra kjernemodus. Resten av minnet skal ikke være aksesserbart, så det skrives til null. Dette er implementert slik:

```
1  static volatile __attribute__ ((aligned (0x4000))) uint32_t page_table[4096];
2  void mmu_init_table(void) {
3          uint32_t base = 0;
4          // initialize page_table
5          // 1024MB - 16MB of kernel memory
6          // each page describes xx00000-xxFFFFF.
7
8      // first page is operating system
9          page_table[base] =     SET_FORMAT_SECTION
10                              | base << SECTION_BASE_ADDRESS_OFFSET
11                              | SECTION_SHAREABLE
12                              | SECTION_ACCESS_PL1_RW_PL0_NONE
13                              | SECTION_OUT_INN_WRITE_BACK__WRITE_ALOC
14                              | SECTION_EXECUTE_ENABLE
15                              | 0 << SECTION_DOMAIN
16                              | SECTION_GLOBAL
17                              | SECTION_NON_SECURE;
18
19     base++;
20     // the next ones are for processes. they are not loaded yet,
21     // so should be only accessable by PL1
22         for (; base < 1024-16; base++) {
23             page_table[base] = 0;
24         }
25
26         // 16 MB peripherals at 0x3F000000-3fffffff
27         for (; base < 1024; base++) {
28             page_table[base] =       SET_FORMAT_SECTION
29                                    | base << SECTION_BASE_ADDRESS_OFFSET
30                                    | SECTION_SHAREABLE
```

```
31                                      | SECTION_ACCESS_PL1_RW_PL0_NONE
32                                      | SECTION_DEVICE_SHAREABLE
33                                      | SECTION_EXECUTE_NEVER
34                                      | 0 << SECTION_DOMAIN
35                                      | SECTION_GLOBAL
36                                      | SECTION_NON_SECURE;
37          }
38          // we have now reached 0x400000000 which is 1 GB, the size of RPI2 memory
39          // which means that the rest of memory should be invalid
40          for (; base < 4096; base++) {
41              page_table[base] = 0;
42          }
43          return;
44  }
```

6.3 Skru på MMU

Når oversettingstabellen er bygget kan man gå igang med å skrive til registre som konfigurere MMU-en. Først skriver den til ACTLR. Den har et viktig bit, SMP (bit 6). SMP er høy vil forespørslene til prosessoren være koherente. Denne må settes høy før MMU-en kan skrus på. Det neste som skjer er at domain null (som vi brukte i oversettingstabellen) blir satt til å være i client. Det betyr at aksesstillatelsene som blir brukt i oversettingstabellen er gjeldende. Det neste som skjer er å si til MMU-en hvor denne oversettingstabellen ligger ved å skrive til TTBR0. Dette registeret konfigurerer også hurtigbufferinstillingene til translation lookaside buffer, som overgangstabellen kommer til å bli lastet opp i.

```
1   void mmu_configure(void) {
2           // set SMP bit in ACTLR. MUST be done before the MMU is enabled.
3           uint32_t auxctrl;
4           __asm volatile (''mrc p15, 0, %0, c1, c0,  1'' : ''=r'' (auxctrl));
5           auxctrl |= 1 << SMP_ENABLE_COHERENT_REQUESTS;
6           __asm volatile (''mcr p15, 0, %0, c1, c0,  1'' :: ''r'' (auxctrl));
7
8       // set domain 0 to be domain manager
9       uint32_t dacr = (DOMAIN_MANAGER << 0);
10          __asm volatile (''mcr     p15, 0, %0, c3, c0, 0'' :: ''r'' (dacr));
11
12          // configure TTBRC to always use TTBR0
13          // the reset value is the correct one, but it can't hurt to be safe!
14          __asm volatile (''mcr p15, 0, %0, c2, c0, 2'' :: ''r'' (0));
15
16      // configure the TTBR0
17          uint32_t ttb_descr =        (unsigned)&page_table
18                                  |   TTB_SHAREABLE
19                                  |   TTB_REGION_OUT_WRITEBACK_WRITE_ALLOC_CACHEABLE
20                                  |   TTB_INNER_WRITETHROUGH_WRITE_ALLOC_CACHEABLE;
21
```

```
22          __asm volatile (''mcr p15, 0, %0, c2, c0, 0''
23                          :: ''r'' (ttb_descr));
24          barrier_instruction();
25  }
```

MMU-en startes i cpu_control_config. Denne skriver til SCTLR (system control register) som gjør noe system konfigurering. Den skrur blant annet på MMU-en, hurtigbufferen og branch-prediction. Ved å skru på hurtigbufferen og branch-prediction vil dataaksesser ta mye mindre tid og systemet vil oppleves å kjøre mye raskere.

```
1   void cpu_control_config(void){
2           uint32_t mode =   MMU_ENABLE
3                           | ALIGNMENT_CHECK_ENABLE
4                           | CACHE_DATA_ENABLE
5                           | BARRIERS_CP15_ENABLE
6                           | SWP_SWPB_ENABLE
7                           | BRANCH_PREDICTION_ENABLE
8                           | CACHE_INSTRUCTION_ENABLE
9                           | VECTORS_LOW
10                          | INTERRUPT_VECTORS_STD
11                          | CACHE_PLACEMENT_STRATEGY_PREDICTABLE
12                          | HARDWARE_ACCESS_FLAG_DISABLE
13                          | INTERRUPT_VECTORS_STD
14                          | EXCEPTION_ENDIANNESS_LITTLE
15                          | TEX_REMAP_DISABLE
16                          | ACCESS_FLAG_DISABLE
17                          | THUMB_EXCEPTION_DISABLE ;
18          __asm volatile (''mcr p15, 0, %0, c1, c0, 0'' :: ''r'' (mode) : ''memory'');
19  }
```

6.4 Oppdatere oversettingstabellen

For å oppdaterer oversettingstabellen brukes en funksjon som tar inn en ny oversettelse fra virtuelt til fysisk minne. Den bruker databarrierer for å sikre at TLB er fullstendig oppdatert før prosessoren gjør nye minneaksesser.

```
1   void mmu_remap_section(uint32_t virt, uint32_t physical, uint32_t config_flags){
2       // 1. invalidate old translation table entry.
3       page_table[virt >> SECTION_BASE_ADDRESS_OFFSET] = 0;
4       // 2. execute dsb.
5       barrier_data_sync();
6       // 3. invalidate the translation table
7       //      with a broadcast TLB invalidation instruction.
8       // Invlaidate unified TLB by writing to TLBIALL
9       __asm volatile (''mcr p15, 0, %0, c8, c7, 0'' :: ''r'' (0));
10      // 4. write new entry.
11      page_table[virt >> SECTION_BASE_ADDRESS_OFFSET] = config_flags
12          |( (physical >> SECTION_BASE_ADDRESS_OFFSET) << SECTION_BASE_ADDRESS_OFFSET);
```

```
13      // 5. data barrier
14      barrier_data_sync();
15      __asm volatile (''mcr p15, 0, %0, c8, c7, 0'' :: ''r'' (0));
16      barrier_data_sync();
17  }
```

7 Minnet

Dette kapittelet omhandler hvordan KybOS organiserer minnet. KybOS må holde styr på hvor den har plassert prosesser i det fysiske minnet. En viktig funksjonalitet er hvordan operativsystemet bruker MMU-en.

I dette kapittelet introduseres disse funksjonene og variablene:

memory_slots _physical Kartet over alle minnesegmentene i det fysiske minne. Hver indeks svarer til 1 MB minne, fordi KybOS bruker virtuelt minne med 1 MB minnesegmenter som beskrevet i kapittel 6.

memory_init initialiserer minnekartet.

memory_slot_get Som beskriver hvordan KybOS tildeler og reservere

virtual_memory_slot_get gir ett nytt segment med virtuel minne til en prosess.

memory_add_mapping gir en ny oversettelse for virtuelt til fysisk minne til en prosess.

memory_remove_mapping Fjerner en gitt oversettelse.

memory_map Gir prosesser tilgang til fysisk minne.

memory_perform_process_mapping Skriver alle minneoversettelsene som prosessene har knyttet til seg til oversettingstabellen.

7.1 kart over fysisk minne

KybOS har et kart over det fysiske minnet som er 1 GB stort. Det fysiske minet er delt inn i 1 MB store områder (på grunn av hvordan MMU-en er konfigurert), som kan enten være ledig, opptatt eller reservert som innput/output minne. Kartet initialiseres av funksjonen memory_init. Første MB er opptatt av kjernen. Resten er fri bortsett fra et området på slutten av minnet som brukes som input/output minne. Dette betyr at KybOS har plass 1007 prosesser i minnet samtidig.

```
1 typedef enum {
2     MEM_VACANT,
3     MEM_OCCUPIED,
4     MEM_IO,
5 } mem_slot_t;
6
7 /*
```

```
8  * using 1 MB sections
9  * this means that there are 1 GB/1MB = 1024
10 * slots of physical memory
11 */
12 static int8_t memory_slots_physical[PHYS_MEM_SIZE/MMU_PAGE_SIZE];
13
14 void memory_init(void){
15     // file secions of memory that are filled by OS
16     memory_slots_physical[0] = MEM_OCCUPIED;          // kernel and interrupts are here!
17     uint32_t i = PERIPHERAL_BASE;
18     for(i = PERIPHERAL_BASE/MMU_PAGE_SIZE; i < PHYS_MEM_SIZE/MMU_PAGE_SIZE; i++){
19         memory_slots_physical[i] = MEM_IO;      // peripherals are memory mapped here
20     }
21 }
```

7.1.1 Fordele fysisk minne

Når en prosess skal initialiseres eller trenger mer minne brukes denne funksjonen for å sikre at minnet som prosessen får blir ikke blir brukt til noe annet og at minnesegmentet blir reservert. Den itererer over kartet og returnerer når den finner en ledig plass.

```
1  /*
2   * returns a pointer to start of a 1 MB section of memory
3   * and sets that section as occupied
4   */
5  void* memory_slot_get(void){
6      uint32_t i = 0;
7      while( memory_slots_physical[i] != MEM_VACANT){
8          i++;
9          if( i > (PHYS_MEM_SIZE/MMU_PAGE_SIZE)){ return NULL;}
10     }
11     memory_slots_physical[i] = MEM_OCCUPIED;
12     return (void*)(i << SECTION_BASE_ADDRESS_OFFSET);
13 }
```

7.2 Gi en prosess mer virtuelt minne

Virtual_memory_slot_get ir en oversettelse fra et virtuelt minnesegment til et fysiske segment. Denne legges i en lenket liste i prosessens PCB. Den lager først et buffer med et calloc kall, som skal representere det virtuelle minnet sett fra prosessen. Deretter initialiserer den bufferen ved å skrive hvor den vet allerede er opptatt. Den første segmentet er opptatt av kjernen. Dette skriver på linje 14. Deretter iterere den gjennom den lenkede listen som inneholder alle virtuelle til fysiske oversettelser som prosessen har, og skrive den i kartet. Dette gjøres på linnje 19 til 22. Deretter søker funksjonen gjennom kartet til den finner en ledig plass. Denne adressen returneres.

```
1  typedef struct PCB{
2      ...
3      mem_mapping_t *mem_next;
4      ...
5  } PCB_t;
6
```

```
7  /*
8   * returns a virtual memory address that can be used for additional mapping of memory
9   * useful when drivers need access to memory mapped io
10  */
11 void* virtual_memory_slot_get(process_id_t id){
12     int i = 0;
13     int8_t *is_occupied = (int8_t*)calloc(( VIRT_MEM_SIZE/MMU_PAGE_SIZE), sizeof(int8_t));
14     is_occupied[0] = MEM_OCCUPIED; // 0 is reseved for kernel
15
16     PCB_t *pcb = pcb_get(id);
17     mem_mapping_t* node = pcb->mem_next;
18     // search for memory that is already mapped for the process
19     while(node){
20         is_occupied[ (node->virtual_address >> SECTION_BASE_ADDRESS_OFFSET) ] = MEM_OCCUPIED;
21         node = node->mem_next;
22     }
23     // select a slot
24     for(i = 0; i < VIRT_MEM_SIZE/MMU_PAGE_SIZE; i++){
25         if(is_occupied[i] == MEM_VACANT){
26             free(is_occupied);
27             return (void*)(i << SECTION_BASE_ADDRESS_OFFSET);
28         }
29     }
30     return NULL;
31 }
32
33 int memory_add_mapping(process_id_t id, uint32_t virtual, uint32_t physical){
34     // get new node
35     mem_mapping_t* node = memory_map_get(virtual, physical);
36     if(node == NULL){    return -1;}
37     PCB_t* pcb = pcb_get(id);
38     mem_mapping_t* it = pcb->mem_next
39     pcb->mem_next = node
40     node->mem_next = it;
41     return 1;
42 }
```

Og for å fjerne en slik node gjøres slik. Funksjonen itererer itererer gjennom den den lenkede listen til den finner noden som skal slettes, og sletter den:

```
1  int memory_remove_mapping(process_id_t id, uint32_t virtual, uint32_t physical){
2      PCB_t* pcb = pcb_get(id);
3      mem_mapping_t* del;
4      mem_mapping_t** it = &pcb->mem_next;
5      while(*it){
6          if( (*it)->virtual_address == virtual && (*it)->physical_address){
7              del = *it;
8              *it = (*it)->mem_next;
9              free(del);
10             return 1;
11         }
12         it = &(*it)->mem_next;
13     }
```

```
14      return 0;
15  }
```

7.3 Memory_map

Når prosesser skal skrive til et sted i det fysiske minnet må prosessen si ifra til operativsystemet. Grunnen til dette er at prosessen kjører i virtuelt minne og aksesserer ikke nødvendigvis det minne prosessen tror den aksesserer. Prosessen har heller ikke nødvendigvis tilatelse til å skrive til denne adresse, noe som vil resultere i at minneaksessen gir ett avbrudd. Dette er nødvendig når for eksempel drivere skal skrive til input-output registre som ligger i minnet.

Løsningen på dette er MMAP systemkallet. Dette systemkallet kaller memory_map. Først må funksjonen sjekke om den allerede har tilgang på minnesegmentet som adress peker til ved å iterere gjennom alle nodene som beskriver alle virtuellt til fysisk oversettelsene som prosessen har knyttet til seg. dette gjøres på linje 2 til 16. Om det ikke finnes må kjernen gi prosessen et til segment med virtuelt minne og lage en ny oversettelse som går fra dette minnesegmentet til det fysiske minnesegmentet som prosessen ønsker å få tilgang til.

```
1  scheduling_type_t memory_map(void* retval, uint32_t address, process_id_t id){
2      // check if region already is mapped
3      PCB_t* pcb = pcb_get(id);
4      mem_mapping_t *node = pcb->mem_next;
5      uint32_t ret;
6      int found  = 0;
7      while( node){
8          if( ((uint32_t)address >> SECTION_BASE_ADDRESS_OFFSET)
9                  == (node->physical_address >> SECTION_BASE_ADDRESS_OFFSET)){
10             //  found another mapping that exists. use this one
11             ret = node->virtual_address;
12             found = 1;
13             break;
14         }
15         node = node->mem_next;
16     }
17     // if not in existing mapping, create a new one
18     if( found != 1){
19         ret = (uint32_t)virtual_memory_slot_get(id);
20         memory_add_mapping(id, ret, address);
21     }
22     // ret now contains the base address
23     // adjust for the rest!
24     ret += address % MMU_PAGE_SIZE;
25     *(uint32_t*)retval = ret;
26     return NO_RESCHEDULE;
27 }
```

7.4 Bytte virtuelt minne under kontekst bytte

Brukes under et kontekst bytte da prosesser har forskjellig mappinger. Denne funksjonen itererer seg gjennom nodene som inneholder oversettelser, og bruker mmu_remap_section, som ble beskrevet i 6.4. Funksjonen gjør det mulig å skrive og lese fra både PL1 og PL2. Om minnet er input-output minne vil det brukes ikke brukes hurtigbuffer.

```
1  int memory_perform_process_mapping(process_id_t id){
2      PCB_t *pcb = pcb_get(id);
3      mem_mapping_t *node = pcb->mem_next;
4      while(node){
5          // check what type of memory it is. if it is IO memory
6          // then dont use the cache!!
7          if( memory_slots_physical[
8                  (node->physical_address >> SECTION_BASE_ADDRESS_OFFSET)] == MEM_IO){
9
10                 mmu_remap_section(       node->virtual_address,
11                                     node->physical_address,
12                                     SET_FORMAT_SECTION
13                                       | SECTION_SHAREABLE
14                                       | SECTION_ACCESS_PL1_RW_PL0_RW
15                                       | SECTION_DEVICE_SHAREABLE
16                                       | SECTION_EXECUTE_NEVER
17                                       | 0 << SECTION_DOMAIN
18                                       | SECTION_GLOBAL
19                                       | SECTION_NON_SECURE);
20         }else{  // if not io memory, use the cache!!
21             mmu_remap_section(  node->virtual_address,
22                                 node->physical_address,
23                                     SET_FORMAT_SECTION
24                                   | SECTION_SHAREABLE
25                                   | SECTION_ACCESS_PL1_RW_PL0_RW
26                                   | SECTION_OUT_INN_WRITE_BACK__WRITE_ALOC
27                                   | SECTION_EXECUTE_ENABLE
28                                   | 0 << SECTION_DOMAIN // set the domain of this section to 0
29                                   | SECTION_GLOBAL
30                                   | SECTION_NON_SECURE);
31         }
32         node = node->mem_next;
33     }
34     return 1;
35 }
36
37 int memory_perform_process_unmapping(process_id_t id){
38     PCB_t *pcb = pcb_get(id);
39     mem_mapping_t *node = pcb->mem_next;
40     while(node){
41         mmu_remap_section(  node->virtual_address, 0, 0);
42         node = node->mem_next;
43     }
44     return 1;
45 }
```

8 Prosesser og multitasking

Prosesser er en svært viktig abstraksjon i et operativsystem som gjør at programmer kan kjøre uavhengig av hverandre. Prosesser i KybOS er implementert ved å laste et program inn i det fysiske minnet, i en 1 MB seksjon. Kjernen lagrer informasjon om prosessen i en Prosess Controll Block. Når prosessen skal kjøres utfører kjernen et kontekst bytte til denne prosessen, og lar prosessen kjøre. Kjernen får kontrollen tilbake ved avbrudd, som kan være generert av et systemkall eller en enhet, for eksempel en klokke. Både et systemkall og en avbrudd kan føre til at kjernen må bytte kontekst.

I dette kapittelet vises hvordan KybOS implementerer prosesser og hvordan den utfører et kontekstbytte. Prosesser er også implementert med tanke på at det skal være mulig for operativsystemet å flytte på. For å gjøre det må operativsystemet vite hvor i minnet det ligger, hvor stakken er og hvor lang heapen er.

I dette kapittelet introduseres disse funksjonene og datatypene som kjernen bruker. Sammenhengen mellom disse prosessene kan ses i figur 8.1. Deretter presenteres funksjoner som brukes til å starte og stoppe prosesser:

PCB som er en samling av informasjon som kjernen har om hver prosess

Kontekst stakk som er måten kjernen lagrer registrene i prosessoren ved et kotekst bytte.

build_context_stack Som bygger kontekst stakken.

contekst_switch og contekst_switch _c som utfører kontekst byttet.

pop_context_stakk som popper kontekst stakken til den nye prosessen på slutten av et kontekst bytte.

interrupt_vector og interrupt_vector_c som tar imot IRQ-avbrudd. Den håndterer avbruddet og finner ut om man skal gjøre et kontekst bytte. Deretter returnerer den til prosessen, eller begynner et kotekst bytte.

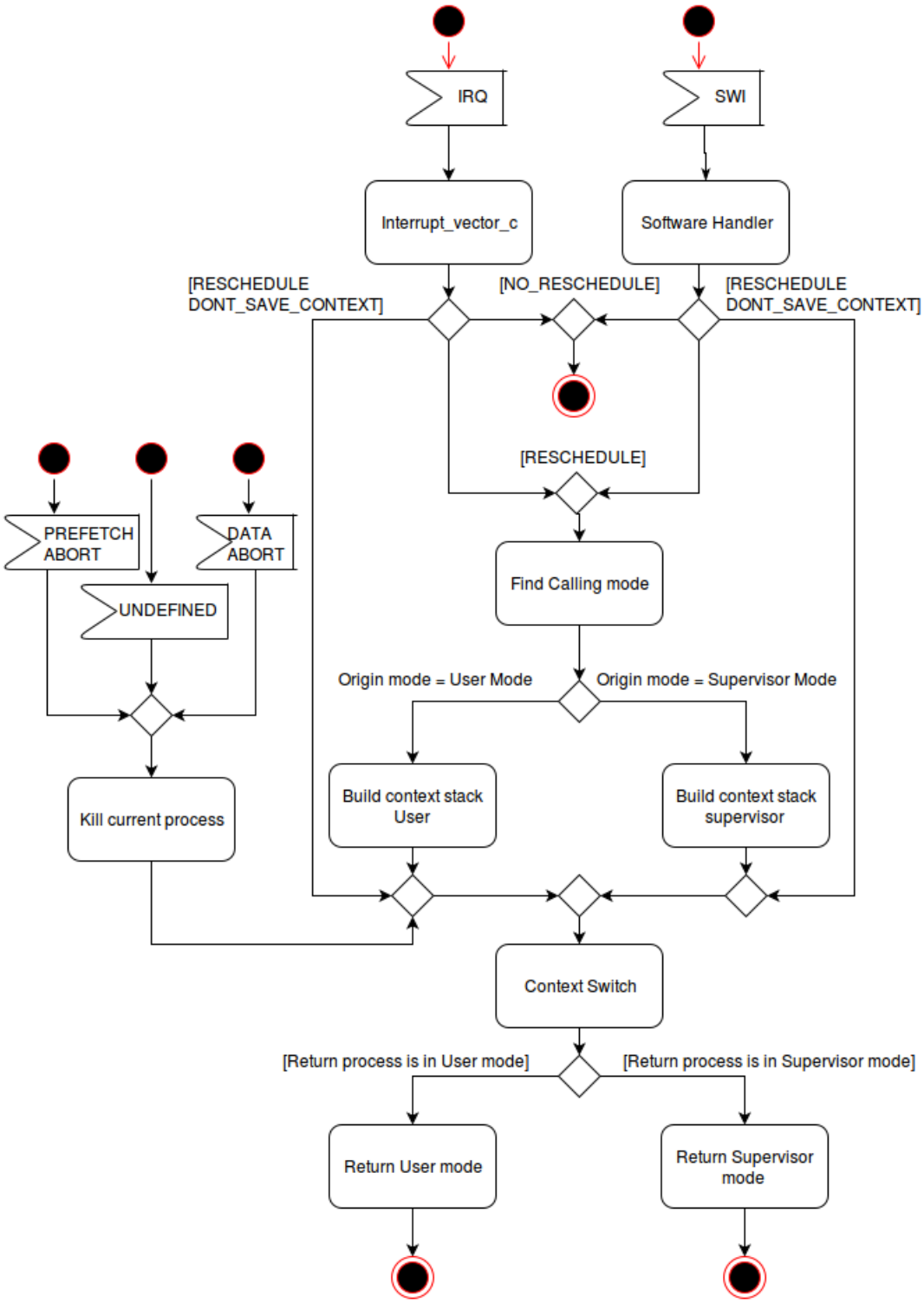
software_interrupt_vector og software_interrupt_c som tar imot systemkall. Gjør ellers det samme som interrupt_vector

prefetch, data og undefined_abort_vector tar imot prefetch-, data- og undefined-avbrudd. De avslutter den nåværende prosessen (fordi den har gjort en handling som genererer et feil-avbrudd) og gjør et kontekst bytte til en annen prosess.

process_spawn starter en ny prosess.

process _load laster en elf fil fra SD-kortet og plasserer den i minnet slik at den kan kjøres som en prosess.

process_kill stopper en prosess.



Figur 8.1: Aktivitetsdiagram av multitasking i KybOS

8.1 Process Control Block

PCB-ene er lagret i en dobbeltlenket liste i kjernen. Den ser slik ut:

```
1  typedef enum{
2      BLOCKED_SENDING,
3      BLOCKED_RECEIVING,
4      READY,
5  } process_state_t;
6
7  typedef struct PCB{
8      process_id_t id;
9      process_state_t state;
10     uint32_t priority;
11     uint32_t stack_start;
12     uint32_t stack_pointer;
13     uint32_t heap_end;
14     uint32_t physical_address;
15     mem_mapping_t *mem_next;
16     struct PCB* next;
17     struct PCB* prev;
18     int is_queued;
19     msg_queue_t msg_queue[NUM_PRIORITIES];
20 } PCB_t;
```

En PCB er søkbar på identitet. Identiteten kan være hva som helst, men en god løsning med tanke på at systemet skal være distribuert er at identiteten inneholder IP-adressen og et nummer. Prosessens tilstand kan være enten BLOCKED_SENDING, BLOCKED_RECEIVING eller READY. Dette er tilstander som brukes av planleggeren. Prosessene har en prioritet. Jo høyere prioritet jo mer vil den bli kjørt på bekostning av andre. KybOS' planlegger kjører alltid den prosessen som har høyest prioritet, så lenge den ikke er blokkert.

Deretter følger viktige variabler med tanke på å holde styr på hvor prosessen ligger i minnet. Stack_start, som peker på hvor i det virtuelle minnet prosessens stakk starter. Stack_pointer inneholder adressen til prosessens nåværende stakk topp. Heap_end inneholder slutten på heapen, og physical_address er starten på prosessen. Ettersom heapen følger direkte etter prosessen ved operativsystem hele minneområdet som prosessen bruker.

Det neste er mem_next. Dette er en linket liste over alle virtuelle til fysiske oversettelser som må gjøres når kjernen bytter kontekst til prosessen. Is_queued brukes av planleggeren for å finne ut om prosessen allerede er i ventelista for å bli kjørt. Til slutt følger en meldingskø som inneholder meldinger som prosessen mottar.

8.2 Å bytte kontekst

Når en ønsker å bytte kontekst må alle registrene lagres i PCB til prosessen. I dette prosjektet ble det valgt å dytte alle registrene på stakken og lagre stakkvariablen i PCB blokken. Denne stakken, kalt kontekst retur stakk, ser ut som i figur 8.1. Denne stakken bygges i samme modus som prosessen

kjører i. Her bygges stakken for en som kjører i brukermodus. For Supervisor modus gjøres det samme, bortsett fra at man går til supervisor modus, og ikke systemmodus.

```
1  build_context_stack_user:
2      // save r0-r12, lr, pc, cpsr on the stack, in that sequence
3      push {r0}
4      mov r0, #(CPSR_MODE_SYSTEM | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
5      msr cpsr_c, r0
6      push {r1-r12}
7      mov r0, #(CPSR_MODE_IRQ | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
8      msr cpsr_c, r0
9      pop {r0}
10     mrs r1, spsr
11     mov r2, lr
12     mov r3, #(CPSR_MODE_SYSTEM | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
13     msr cpsr_c, r3
14     push {r0}
15     push {r1,r2, lr}
16
17     mov r0, sp // save SP so we can use it as parameter in the handler
18
19     mov r1, #(CPSR_MODE_IRQ | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
20     msr cpsr_c, r1
21
22     b context_switch
```

cpsr
pc
lr
r1
r2
...
r11
r12

Tabell 8.1: Kontekstbytte stakk

Deretter må kjernen finne ut hvilken prosess som man skal bytte kontekst til. Dette gjøres i funksjonene contect_switch.

```
1  context_switch:
2      bl context_switch_c // givecontext_switchent, returns with a new sp
3
4      mov r1, sp  // irq mode sp
5      mov sp, r0  // target sp
6      mov r4, r0  // store target sp in r4
7
8  // determine target mode by pop the cpsr
9      pop {r0}
10     push {r0}
11     mov sp, r1  // move irq sp back
12     and r0, r0, #(CPSR_MODE_FILTER)
13     cmp r0, #(CPSR_MODE_USER)
14     beq pop_return_stack_user
15     cmp r0, #(CPSR_MODE_SVR)
16     beq pop_return_stack_svr
```

```
1  uint32_t context_switch_c(uint32_t old_sp){
2      // save old id
3      process_id_t previous_running_process = get_current_running_process();
4      // reschedule for new id
5      reschedule();
6      PCB_t* pcb = pcb_get( previous_running_process);
7      if(pcb != NULL){
```

```
8           // if no errors, save old sp
9           pcb->stack_pointer = old_sp;
10          memory_perform_process_unmapping(previous_running_process);
11          // unmap memory of old process
12      }else{
13          // if a process is killed, the pcb will be NULL here.
14          // if so, we don't know which slots to unmap.
15          // this can be solved. Quick fix: initiate all of the page table again.
16          // not efficient. but works.
17          mmu_init_table();
18          mmu_table_update();
19      }
20
21
22      // load new sp
23      pcb = pcb_get( current_running_process);
24      if (pcb == NULL){
25          uart_puts(''KERN Error: Load process context_switch_c\r\n'');
26          //  if cant load a process stack pointer, return with...
27          //  IDK, zero I guess. Something something something...
28          return 0;
29      }
30      memory_perform_process_mapping(current_running_process);
31          // if no errors, return the stack pointer!
32      return pcb->stack_pointer;
33  }
```

Dette gjøres ved å kjøre funksjonen ”reschedule” som finner prosessen med høyest prioritet som også er i tilstanden ready. Planleggeren er implementert som 64 linkede lister som hver representerer et prioritetsnivå. Dette gir 64 prioritetsnivåer. Etter at ”reschedule” har kjør lagres stakk pekeren til den game prosessen. Den fjerner også gamle verdier som tilhører prosessen i oversettelsestabellen til MMU-en. Deretter henter kjernen stakk pekeren til den nye prosessen, skriver nye verdier til oversettelsestabellen slik at prosessen har tilgang til det virtuelle minner det trenger, og returnerer med den nye stakk pekeren. Kjernen bruker nå den nye stakk pekeren til å poppe den nye prosessens kontekst stakk. Det siste context_switch gjør er å finne ut hvilken modus den nye prosessen kjører i. Dette gjøres i linje 11 til 16. Deretter kaller context_switch den riktige funksjonen for å poppe kontekst stakken. For brukermodus ser ser den slik ut.

```
1   pop_return_stack_user:
2       // build the return stack
3       mov r0, #(CPSR_MODE_SYSTEM | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
4       msr cpsr_c, r0
5       mov sp, r4
6       pop {r0-r2}     // spsr, pc, lr
7       mov lr, r2
8       mov r3, #(CPSR_MODE_IRQ | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
9       msr cpsr_c, r3
10      push {r1} // pc
11      msr spsr, r0 // cpsr
```

```
12      mov r4, #(CPSR_MODE_SYSTEM | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
13      msr cpsr_c, r4
14      pop {r0-r12}
15      push {r0}
16      mov r0, #(CPSR_MODE_IRQ | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
17      msr cpsr_c, r0
18      push {r1-r12}
19      mov r0, #(CPSR_MODE_SYSTEM | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
20      msr cpsr_c, r0
21      pop {r0}
22      mov r1, #(CPSR_MODE_IRQ | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
23      msr cpsr_c, r1
24      push {r0}
25      ldm sp!, {r0-r12, pc}^
```

Hatten på slutten av linje 25 sier at dette er en instruksjon som skal returnere fra et avbrudd. Det som skjer er at Verdien av SPSR (som er den lagrede verdien av CPSR) lastes tilbake til CPSR. Dette betyr at man også bytter modus til verdien som er i SPSR. Registrene i registerlista refererer til registrene til modusen i SPSR, dermed vil link registerets verdi som ble dyttet på stakken bli lastet opp i programpekeren. Etter at instruksjonen på linje 25 er eksekvert vil prosessen være startet opp igjen, og kontekst byttet er utført.

8.3 Multitasking

Ved å bytte kontekst hurtig mellom flere prosesser, å multitaske prosesser, gir en illusjon om at prosessene kjører parallelt. Prosessene skal selv ikke merke at de blir byttet ut og inn. Dette er en essensiell oppgave for kjernen. Disse kontekst byttene har sin kilde i at et avbrudd skjer. Dette kan enten være enheter som genererer et avbrudd (for eksempel en klokke), et systemkall, eller en feil som MMU-en oppdager (for eksempel at en prosess prøver å aksessere til en minneadresse den ikke har lov til). Kjernen må behandle disse avbruddene og finne ut om kjernen skal bytte kontekst til en annen prosess. Et aktivitetsdiagram for kjernens multitasking finnes i figur 8.1. Her følger implementasjonen av hvordan KybOS multitasker sine prosesser.

Avbrudd	offset
Undefined	+4
Software Interrupt	0
Prefetch abort	+4
Data abort	+8
Interrupt	+4
Fast interrupt	+4

Tabell 8.2: Link registerets offset

Et enhetsavbrudd vil gi et hopp til interrupt-handleren: interrupt_vector. Det første som gjøres er å finne ut hvilken verdi som programpekeren hadde rett før avbruddet. På grunn av at prosessoren finner avbruddet først *etter* at prosessoren har inkrementert programpekern vil det være et offset på 4, som vist i tabell 8.2. Dette justeres for på linje 2. Deretter kalles funksjonen interrupt_vector_c. Denne behandler avbruddet, og returnerer med en verdi som avgjør om det skal gjøres et kontekst bytte.

Om funksjonen returnerer med verdien 0, så skal det ikke gjøres et kontekst bytte, man kan ganske enkelt returnere til prosessen som ble avbrutt, ved linje 6. Om verdien er 1 skal det gjøres et kontekst bytte til en annen prosess. Da må kjernen bruke find_origin_context_switch for å finne ut hvilken

modus som den gamle prosessen kjørte i, slik at man kan bruke riktig funksjon til bygge kontekst stakken. Om verdien er 2 skal det også gjøres et kontekst bytte, men kjernen trenger ikke å lagre den gamle konteksten. Dette kan forkomme om den gamle prosessen ikke lengre eksisterer, på grunn av et exit eller kill systemkall. Da hopper man over å bygge kontekst stakken.

```
1  interrupt_vector:
2      sub lr, lr, #4
3      push {r0-r3, ip, lr}
4      bl interrupt_vector_c   // returns with 1 in r0 if to perform context switch
5      cmp r0, #0          // if r0 is zero, do not perform context switch
6      ldmeq sp!, {r0-r3, ip, pc}^
7      // if not, branch to context switch with correct mode
8      cmp r0, #2
9      pop {r0-r3, ip, lr}
10     // if r0 == 2, branch to a context switch, but dont build a return stack
11     moveq sp, #(STACK_IRQ)
12     beq context_switch_no_build_stack
13     // if interrupt vector returns with 1, do a context switch
14     // and build a return stack
15 find_origin_context_switch:
16     push {r0}
17     mrs r0, spsr
18     and r0, r0, #(CPSR_MODE_FILTER)
19     cmp r0, #(CPSR_MODE_USER)
20     popeq {r0}
21     beq build_context_stack_user     // perform context switch from user
22     cmp r0, #(CPSR_MODE_SVR)
23     popeq {r0}
24     beq build_context_stack_svr     //else from supervisor
25
26     // stack should already be set to appropriate value
27     // SP given to context_switch can be garbage
28 context_switch_no_build_stack:
29     // move to irq modus
30     mov r1, #(CPSR_MODE_IRQ | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT)
31     msr cpsr_c, r2
32     b context_switch
```

Om avbruddet kommer fra et systemkall vill det genereres et hopp til software_interrupt_vector. Den gjør mye av det samme som interrupt_vector. Først sikrer den seg at stakken er kjernestakken i supervisor modus (om en prosess kjører i kjernemodus og gjør et systemkall vil ikke stakken bytes ut med en stakk som ligger i skyggeregisteret, fordi prosessoren allereder er i supervisor mode). Deretter kalles software_interrupt_vector, som behandler systemkallet. Funksjonen returnerer med en verdi som avgjør om kjernen skal gjøre et kontekst bytte eller ikke, på akkurat somme måte som interrupt_vector.

```
1  software_interrupt_vector:
2      // use OS svr stack
3      push {r4}   // store r4 for extra space
4      mov r4, sp  // store current process stack in r4
5      mov sp, #(STACK_SVR)
6
```

```
7       // call the software interrupt vector in c
8       push {ip, lr}
9       bl software_interrupt_vector_c
10      pop {ip, lr}
11      mov sp, r4  // move current process stack back into sp
12      pop {r4}    // retrieve r4 that was stored earlier
13      // there is now zero stack used
14      // check if to context switch
15      // software_interrupt_vector_c returns with 1 or 0.
16      // 1 means to context switch!
17      // 2 means to context switch without saving the context
18      cmp r0, #1
19      beq software_context_switch
20      cmp r0, #2
21      moveq sp, #(STACK_SVR)
22      beq context_switch_no_build_stack
23      // if not, return
24      push {lr}
25      ldm sp!,{pc}^
26
27
28      //  Called when a user thread calls a system call that  generates a reschedule
29  software_context_switch:
30      push {r0-r2}
31      mrs r0, spsr
32      mov r1, lr
33      mov r2, #(CPSR_MODE_IRQ | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
34      msr cpsr_c, r2
35      msr spsr, r0
36      mov lr, r1
37      push {r3}
38      mov r2, #(CPSR_MODE_SVR | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
39      msr cpsr_c, r2
40      pop {r0-r2}
41      mov r3, #(CPSR_MODE_IRQ | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT )
42      msr cpsr_c, r3
43      pop {r3}
44      b find_origin_context_switch
```

Avbruddene som genereres når en feil forekommer er prefetch abort, data abort og undefined. Disse vil gi stakk pekeren og program pekeren som parametre til en C-funksjon, som printer ut hvor feilen forkom og vil drepe prosessen som genererte feilen. Dette er koden for en prefetch abort. De andre to feilenen håndteres på samme måte.

```
1   prefetch_abort_vector:
2       sub lr, lr, #4
3       push {r0-r4, ip, lr}
4       mov r0, lr
5       mrs r3, cpsr
6       mrs r2, spsr
7       and r4, r2, #(CPSR_MODE_FILTER)
8
9       cmp r4, #(CPSR_MODE_USER)
```

```
10      // if from user mode, go back to system mode!
11      msreq cpsr_c, #(CPSR_MODE_SYSTEM | CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT)
12      orrne r4, #(CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT)
13      msrne cpsr_c, r4
14      mov r1, sp
15      msr cpsr_c, r3
16      mvn r4, #(CPSR_IRQ_INHIBIT | CPSR_FIQ_INHIBIT)
17
18      // os prefetch_abort_vector_c(link_reg, stack_pointer)
19      bl prefetch_abort_vector_c
20      mov sp, #(STACK_ABORT)
21      b context_switch_no_build_stack
22      // ldm sp! {r0-r4, ip, pc}^
```

```
1  void prefetch_abort_vector_c( uint32_t origin, uint32_t stack ){
2      uart_puts(''prefetch abort from: '');
3      uart_put_uint32_t(origin, 16);
4      uart_puts('' with stack: '');
5      uart_put_uint32_t(stack, 16);
6      uart_puts(''\r\n'');
7      process_kill( get_current_running_process() );
8  }
```

Til de høynive C-funksjonenen interrupt_vector_c er enkle funksjoner som finner ut hvilket avbrudd eller systemkall som har skjedd, og kaller den riktige funksjonen for å håndtere avbruddet.

```
1  uint32_t interrupt_vector_c(void){
2  //  find irq source
3      irq_controller_t *irq_flags = irq_controller_get();
4      //  timer
5      if( irq_flags->IRQ_basic_pending & ARM_TIMER_IRQ ){
6          // Do all timer things
7          // message timer irq driver
8          ipc_kernel_send(NULL, 0, driver_irq_get(DRIVER_TIMER));
9          return time_handler();
10     }
11     // uart
12     if ( irq_flags->IRQ_pending_2 & UART_IRQ){
13         // Do all uart things
14         ipc_kernel_send(NULL, 0, driver_irq_get(DRIVER_UART));
15         return uart_handler();
16     }
17
18     return 0;
19 }
20
21 scheduling_type_t software_interrupt_vector_c(
22             void* arg0,
23             void* arg1,
24             void* arg2,
```

```
25              void* arg3){
26      switch( (system_call_t)arg0) {
27          case IPC_SEND:
28          // system_send(void* payload, uint32_t size, process_id_t coid)
29          return system_send(arg1, (ipc_msg_config_t*)arg2);
30          break;
31          case IPC_SEND_DRIVER:
32          // send a msg to driver
33          return system_send_driver(arg1, (ipc_msg_config_driver_t*)arg2);
34          break;
35          case IPC_RECV:
36          // system_receive(ipc_msg_t *recv_msg, uint32_t size, int* success)
37          return system_receive(arg1, (uint32_t)arg2, (int*)arg3);
38          break;
39          case DUMMY:
40          uart_puts(''DUMMY CALL!!!\r\n'');
41          return NO_RESCHEDULE;
42          break;
43          case YIELD:
44          return RESCHEDULE;
45          break;
46          case EXIT:
47          return process_kill( get_current_running_process() );
48          break;
49          case KILL:
50          return process_kill( *(process_id_t*)arg1);
51          break;
52          case MMAP:
53          // void* memory_map(void' location);
54          return memory_map(arg1, (uint32_t)arg2, get_current_running_process());
55          break;
56          case DRIVER_REGISTER:
57          // register a new driver
58          return driver_register(get_current_running_process(),
59                          (char*)arg1, (int*)arg2);
60          break;
61          case SPAWN:
62          return process_spawn( (spawn_args_t*)arg1);
63          break;
64          case SRBK:
65          return memory_srbk( (int)arg1, get_current_running_process());
66          break;
67      }
68      uart_puts(''Software irq vector c: did not find source of exception!\r\n'');
69      return NO_RESCHEDULE;
70  }
```

8.4 Starte og stoppe prosesser

Grensesnittet for å starte og stoppe prosesser er systemkallene spawn, exit og kill.

8.4.1 Å laste en prosess fra kjernen

Når operativsystemet startes opp eksisterer bare kjernen i minnet. For å starte opp hele operativsystemet må også noen prosesser startes, for eksempel drivere for enheter. Disse prosessene ligger på filer i SD-kortet. Ideelt sett skal alt av enheter styres av drivere som kjøres som prosesser, men det kreves en driver for å laste inn prosesser. Dette er et lite "høna eller egget" problem. Løsningen må være at kjernen har en driver for å laste opp filer til minnet.

process_load er funksjonen som kjernen benytter til å laste inn prosesser. Den laster først opp en fil og plasserer den i en buffer. Under oppstart, før en driver-prosess er startet, må kjernen bruke en intern SD-driver som er kompilert med i kjernen. Etter oppstart burde kjernen bruke en driver istedet, men den er ikke blitt implementert. Så leser den bestemte adresser for å tolke elf-fil formatet. Elf filer er et format som brukes i Unix-verdenen for eksekverbare filer. Denne [5] websiden har vært viktig for å forstå hvordan ELF filer fungerer. Om funksjonen finner en feil avbryter funksjonen og returnerer en feilverdi.

ELF filer er et ganske enkelt format. I starten av filen finnes det en elf-header som inneholder noe informasjon som identifiserer seg som en elf fil, og noe om hvilken arkitektur den er kompilert for. Viktige verdier her er å finne program-headeren, som inneholder informasjon om programmet som skal lastes inn som en prosess. Denne headeren har informasjon om hvor datasegmentet ligger i minnet, hvor filen må lastes inn i virtuelt minne og størrelsen.

Om alle verdiene i ELF headeren er korrekte begynner funksjonen å lage en prosess. Den allokerer en PCB, allokerer en fysisk minne-seksjon. Deretter brukes MMU-en til å gjøre dette minne tilgjengelig, og kopierer program-minnet til minneadressen den er linket til når filen ble kompilert. Dette leses i program_header. Det neste er å initialisere stakken, legge PCB-en i PCB lista og angi fysisk-vituell oversettelse som trengs ved kontekst bytter.

```
1  // 32bit version
2  struct elf_header{
3      uint8_t magic_number;     // should be 0x7f
4      char magic_string[3];     // should be ''ELF''
5      uint8_t word_size;        // 1 == 32bit 2 = 64bit
6      uint8_t endianess;        // 1 == little endiand, 2 == big
7      uint8_t elf_ver;
8      uint8_t OS_ABI;
9      uint8_t reserved[8];
10     uint16_t type;   // 1==relocatable, 2==executable, 3==shared, 4==core
11     uint16_t instruction_set;
12     uint32_t elf_ver2;
13     uint32_t program_entry;
14     uint32_t program_header;
15     uint32_t section_header;
16     uint32_t flags;
17     uint16_t header_size;
```

```
18      uint16_t prog_entry_size;
19      uint16_t prog_entry_num;
20      uint16_t sec_entry_size;
21      uint16_t sec_entry_num;
22      uint16_t sec_name_index;
23  };
24
25  // 32bit version
26  struct program_header{
27      uint32_t segment_type;
28      uint32_t p_offset;      // where segment lies in the file
29      uint32_t p_vadder;      // where it should be put in virtual memory
30      uint32_t undefined;
31      uint32_t p_filesz;      // size of segment in file
32      uint32_t p_memsz;       // size of segment in memory
33      uint32_t flags;
34      uint32_t alignment;
35  };
36
37  // 32bit
38  struct section_header{
39      uint32_t name;
40      uint32_t type;
41      uint32_t flags;
42      uint32_t address;
43      uint32_t offset;
44      uint32_t size;
45  };
46
47  int process_load(const char* file_path, size_t priority,
48                               int mode, process_id_t id){
49      // read first segment of the file.
50      // this contains the elf header and program header
51
52      if( ( pcb_id_compare(id, NULL_ID)) || pcb_get(id) != NULL){
53          uart_puts(''Process load: ID is reserved\r\n'');
54          return -1;
55      }
56      uint8_t *buf = (uint8_t*)malloc(0x20000);
57      int ret = fs_get()->fs_load(file_path, buf, 0x20000);
58      if(ret == -1){
59          uart_puts(''Process load: file not valid\r\n'');
60          free(buf);
61          return -1;
62      }
63      struct elf_header *myheader = (struct elf_header*)buf;
64
65      // read elf header
66      if(( myheader->magic_number != ELF_MAGIC_NUM)
```

```
67          || strncmp(myheader->magic_string, elf_magic_string, 3)){
68          uart_puts(''Process load: file not a elf file\r\n'');
69          free(buf);
70          return -1;
71      }
72      if( myheader->word_size != 1){
73          uart_puts(''Process load: elf file not 32 bit\r\n'');
74          free(buf);
75          return -1;
76      }
77      if( myheader->type != 2){
78          uart_puts(''Process load: file not executable\r\n'');
79          free(buf);
80          return -1;
81      }
82      if( myheader->instruction_set != INSTR_SET_ARM ){
83          uart_puts(''Process load The instruction set
84          is not ARM. Try to use another compiler\r\n'');
85          free(buf);
86          return -1;
87      }
88      // the new process should go in here!
89      void* dest = memory_slot_get();
90
91      struct program_header *prog_header =
92                  (struct program_header*)&buf[myheader->program_header];
93  //    struct section_header *sect_header =
94                  (struct section_header*)&buf[myheader->section_header];
95
96      PCB_t pcb = {    .id = id,
97                       .state = READY,
98                       .priority = priority,
99                       .physical_address = (uint32_t)dest,
100     };
101     // save the old mapping for p_vadder
102     uint32_t old_mapping = mmu_get_mapping(prog_header->p_vadder);
103
104     mmu_remap_section(  prog_header->p_vadder,
105                         (uint32_t)dest,
106                         SET_FORMAT_SECTION
107                       | SECTION_SHAREABLE
108                       | SECTION_ACCESS_PL1_RW_PL0_NONE
109                       | SECTION_OUT_INN_WRITE_BACK__WRITE_ALOC
110                       | SECTION_EXECUTE_ENABLE
111                       | 0 << SECTION_DOMAIN // set the domain of this section to 0
112                       | SECTION_GLOBAL
113                       | SECTION_NON_SECURE);
114
115     memset((void*)prog_header->p_vadder, 0, prog_header->p_memsz);
```

```
116     memcpy((void*)prog_header->p_vadder,
117                     buf + prog_header->p_offset , prog_header->p_filesz);
118
119     // make PCB
120     pcb.stack_start = (1 << 20) - 0x1000 + prog_header->p_vadder;
121
122     pcb.stack_pointer = _process_stack_init(
123                     pcb.stack_start, myheader->program_entry, mode);
124
125     pcb_insert(pcb);
126     memory_add_mapping(id, prog_header->p_vadder, (uint32_t)dest);
127     // unmap process now that we are done with it
128     mmu_remap_section(  prog_header->p_vadder, 0, old_mapping);
129     free(buf);
130     return 1;
131 }
```

8.4.2 Spawn

Spawn er et systemkall som tar en path variable til et program på SD kortet, en ID som sier hvilken identitet prosessen skal ha, mode, som sier om proseesn skal kjøre i kjerne eller brukermodus, en prioritet og enkelte flagg som er nyttige for å returnere feilverdier.

Det første den gjør er å finne ut hvilken modus prosessen skal kjøre i, på linjene 13 til 23. Deretter sjekker den om gitt ID eksisterer. Om den eksisterer avslutter funksjonen. Deretter kaller den process_load.

Som beskrevet i delkapittel om process_load, kapittel 8.4.1, bruker kjernen en intern driver til å laste opp filer. Denne driveren må ha tilgang på tidsavbrudd. Derfor må avbrudd skrus på (linje 32) og avbrytende tidsavbrudd skrus av. Dette reverseres selvfølgelig når kjernen er ferdig med å laste opp på linjer(39-40). Dette har den effekten at det ikke blir gjort noen kontekst bytter mens kjernen laster opp filer. Dette er selvfølgelig katastrofalt for sanntidsegenskapene til operativsystemet, men kan utbedres ved å la en prosess-driver ta seg av denne jobben. Denne driveren har altså enda ikke blitt implementert.

```
1  typedef struct{
2      char* path;
3      process_id_t id;
4      spawn_mode_t mode;
5      size_t priority;
6      int flags;
7  }spawn_args_t;
8
9  #define SPAWN_MODE_ERROR        (1 << 0)
10 #define SPAWN_LOAD_ERROR        (1 << 1)
11 #define SPAWN_ID_OCCUPIED       (1 << 2)
12
13 scheduling_type_t process_spawn( spawn_args_t *args){
14     int mode;
```

```
15      process_id_t id = args->id;
16      if(args->mode == SPAWN_USER){
17          mode = CPSR_MODE_USER;
18      }else if(args->mode == SPAWN_SUPERVISOR){
19          mode = CPSR_MODE_SVR;
20      }else{
21          args->flags |= SPAWN_MODE_ERROR;
22          return NO_RESCHEDULE;
23      }
24      if( pcb_get(args->id) != NULL){
25          args->flags |= SPAWN_ID_OCCUPIED;
26          return NO_RESCHEDULE;
27      }
28      // enable interrupts because process load requires time interrupts
29      // should be moved out to a driver-process, but alas, time...
30      // disable rescheduling for time interrupts
31      set_preemptive_timer(0);
32      _enable_interrupts();
33      if( process_load( args->path, args->priority, mode, args->id) != 1){
34          args->flags |= SPAWN_LOAD_ERROR;
35          _disable_interrupts();
36          set_preemptive_timer(1);
37          return RESCHEDULE;
38      }
39      _disable_interrupts();
40      set_preemptive_timer(1);
41      // disable interrupts
42      process_start(id);
43      return RESCHEDULE;
44  }
```

8.5 Exit og Kill

Exit og kill systemkallene kaller begge funksjonen process_kill. Den tømmer meldingskøen, frigir den fysiske minnet, fjerner den som driver om den tidligere har registrert seg som driver, og til slutt fjerne PCB blokken.

```
1   scheduling_type_t process_kill( process_id_t id){
2       // empty msg queue
3       ipc_flush_msg_queue(id);
4       // free memory
5       memory_slot_free( (void*)pcb_get(id)->physical_address);
6       //free driver
7       driver_remove(id);
8       // remove pcb
9       pcb_remove(id);
10      if( pcb_id_compare(id, get_current_running_process()) == 1){
11          return RESCHEDULE_DONT_SAVE_CONTEXT;
```

```
12      } else {
13          return RESCHEDULE;
14      }
15  }
```

9 Kommunikasjon mellom prosesser

Kommunikasjon mellom prosesser (IPC) er en svært viktig grunnleggende service som kjernen må legge til rette for. For å kunne lage et meningsfylt operativsystem så må det være mulig å kommunisere med andre prosesser i applikasjoner, drivere osv. KybOS har tre systemkall for å kommunisere. Send, receive og send_driver. Send har mulighet for å være både synkronisert og usynkronisert.

9.1 Meldingskøen

Hver prosess må ha en kø som hvor meldinger lagres. En god løsning er å legge den i PCB-en, slik at en PCB inneholder en port mot alle meldingene. En melding er en struct med noe metadata rundt en last. Den sender med hvem som sendte meldingen, noen flagg og størrelsen på meldingen.

```
1  typedef struct ipc_msg{
2      process_id_t sender;
3      int flags;
4      size_t payload_size;
5      struct ipc_msg* next;
6      struct ipc_msg* prev;
7      char payload[0];
8  } ipc_msg_t;
9
10 typedef struct Msg_queue{
11     ipc_msg_t* head;
12     ipc_msg_t* tail;
13 } msg_queue_t;
14
15 typedef struct PCB{
16     ...
17     msg_queue_t msg_queue[NUM_PRIORITIES];
18     ...
19 } PCB_t;
```

Her er det lagt til en meldingskø, som er et array av lenkede lister som brukes som køer. Det er et array fordi det gjør det mulig å enkelt implementere det slikt at man alltid tar ut den meldingen med høyest prioritet. Det er det mulig at det er mange meldinger i køen, og man har alltid lyst til å ta ut den meldingen som er fra prosessen med høyest prioritet. Dette sikrer kortest mulig responstid for prosesser med høy prioritet.

9.2 Meldingingsbasert IPC

Det er to grunnleggende systemkall som må implementeres: send og receive. Her er de implementert slik det blir synkron kommunikasjon. Hver av kallene har 2 sider. Først må en prosess gjøre noe forberedende arbeid, deretter gjøre et systemkall, før kjernen gjør resten. Viktige rettelinjer er at alt som endrer ved en PCB og ting som ligger uten for prosessen må gjøres av kjernen. En annen er at en prosess må på ingen måte få tilgang til minnet til en annen prosess, for eksempel ved å bare sende over en peker til et minnesegment. Det er, med andre ord, viktig at meldingssendingen fungerer ved å kopiere et minnesegment fra en prosess, til en annen.

9.2.1 Send

På brukersiden er funksjonen ipc_send definert. Brukeren må gi en prosess identitet som man ønsker å sende til, en peker i minnet til noe man ønsker å send, størrelsen på minnesegmentet, og flagg.

```
1  #define WAITING_SEND            (1 << 0)
2  #define COID_NOT_FOUND          (1 << 1)
3
4  typedef struct ipc_msg_config{
5      process_id_t coid;
6      int flags;
7      size_t size;
8  }ipc_msg_config_t;
9
10 int ipc_send(void* smsg, ipc_msg_config_t *config){
11     _SYSTEM_CALL(IPC_SEND,(void*)smsg, (void*)config, NULL);
12     return 1;
13 }
```

Kjernesiden av kallet er noe mer komplisert. Først putter den meldingen i køen til riktig prosess. Om prosessen ikke eksisterer returnerer funksjonen med en feilverdi og uten å gjøre et kontekst bytte. Deretter sjekker funksjonen om meldingen er sent som asynkront eller synkront. Om synkron oppførsel er ønsket settes sendende prosess' tilstand til BLOCKED_SENDING. Til slutt finner den ut om den motagende prosessen er blokkert og venter på motta en melding. Hvis dette er tilfellet så settes tilstanden til READY og den dytte på planleggerens kø.

Hvis KybOS skal kunne fungere som et distribuert system må kjernen også sjekke om prosessen også eksisterer på en annen node. Dette må gjøres ved å sende en melding til andre noder som denne noden vet om og, om prosessen eksisterer på en annen node, sende meldingen dit. Da er det heller ikke mulig å bruke synkronisert sending, så å sende til andre noder må være asynkron.

```
#define WAITING_SEND          (1 << 0)
#define COID_NOT_FOUND        (1 << 1)
scheduling_type_t system_send(void* payload, ipc_msg_config_t *config){
    if(ipc_msg_enqueue(payload, config->size,
                config->coid, config->flags,
                get_current_running_process()) == -1){
        config->flags |= COID_NOT_FOUND;
```

```
        return NO_RESCHEDULE;
    }
    if(config->flags & WAITING_SEND){
        pcb_get( get_current_running_process() )->state = BLOCKED_SENDING;

    }
    if (pcb_get(config->coid)->state == BLOCKED_RECEIVING){
        pcb_get(config->coid)->state = READY;
        scheduler_enqueue(config->coid);
    }
    return RESCHEDULE;
}
```

Send_driver er svært lik, bortsett fra at i stedet for å sende en identitet sender man et navn, for eksempel ”timer”. Dette er praktisk fordi KybOS kan finne en driver som har registrert seg med det navnet.

```
// user side
int ipc_send_driver(void* smsg, ipc_msg_config_driver_t *config){
    _SYSTEM_CALL(IPC_SEND_DRIVER,(void*)smsg, (void*)config, NULL);
    return 1;
}

//kernel side
scheduling_type_t system_send_driver(void* payload, ipc_msg_config_driver_t *config){
    process_id_t coid = driver_get(config->name);
    if( ipc_msg_enqueue(payload,
                config->size, coid, config->flags,
                get_current_running_process() ) == -1){
        config->flags |= COID_NOT_FOUND;
        return NO_RESCHEDULE;
    }
    if(config->flags & WAITING_SEND){
        pcb_get( get_current_running_process() )->state = BLOCKED_SENDING;
    }
    if (pcb_get(coid)->state == BLOCKED_RECEIVING){
        pcb_get(coid)->state = READY;
        scheduler_enqueue(coid);
    }
    return RESCHEDULE;
}
```

En tredje send, ipc_kernel_send brukes av kjernen når den skal sende meldinger til prosesser. Dette benyttes blant annet når kjernen skal sende meldinger til driverer som skal behandle et enhetsavbrudd. Den ser lik ut de andre send funksjonene, men med små forskjeller. Denne funksjonen har ikke funksjonalitet for å være blokkerende send. Dette er en selvfølgelighet ettersom kjernen ikke kan blokkeres. En annen er at meldingene må sendes med høyeste prioritet slik at disse meldingene blir plukket ut først av meldingskøen.

```
int ipc_kernel_send(void* payload, size_t size, process_id_t coid){
    if(ipc_msg_enqueue(payload, size, coid, 0, NULL_ID) == -1){
        return 0;
    }
    if (pcb_get(coid)->state == BLOCKED_RECEIVING){
        pcb_get(coid)->state = READY;
        scheduler_enqueue(coid);
    }
    return 1;
}
```

9.2.2 Receive

I brukerdelen av receive krever at brukeren skal gi en plass i minnet og en lengde på meldingen som skal mottas. Funksjonen allokerer plass til meldingen på heapen, kaller systemkallet for receive. Om det ikke er noen meldinger i kø-en vil funksjonen kalle yield, og bli "vekket opp" når det kommer en melding. Når funksjon får tak i en melding kopierer den lasten til meldingen over til pekeren brukeren gav til funksjonen. Til slutt deallokerer funksjonen meldingen som ble allokert tidligere i funksjonen, og returnerer med hvilken prosess som sendte meldingen.

```
1  #define QUEUE_EMPTY          (1 << 0)
2  #define BUF_TOO_SMALL        (1 << 1)
3
4  process_id_t ipc_receive(void* rmsg, size_t buf_size, int* flags){
5      process_id_t sender;
6      ipc_msg_t* recv_msg = malloc( sizeof(ipc_msg_t) + buf_size);
7      _SYSTEM_CALL(IPC_RECV, recv_msg, (void*)buf_size, flags );
8      if( *flags & QUEUE_EMPTY ){
9          _SYSTEM_CALL(YIELD, NULL, NULL, NULL);
10         _SYSTEM_CALL(IPC_RECV, recv_msg, (void*)buf_size, flags );
11     }
12     // clear queue empty, should not be visible to user
13     *flags &= ~(QUEUE_EMPTY);
14     memcpy(rmsg, recv_msg->payload, buf_size);
15     sender = recv_msg->sender;
16     free(recv_msg);
17     return sender;
18 }
```

I kjernedelen av å receive prøver kjernen å ta ut en melding. Om meldingskøen er tom vil funksjonen returnere med en feilmelding og sette prosesstilstanden til blokkert. Om meldingskøen ikke er tom vil kjernen kopiere meldingen fra kjernen sin heap over til prosessen. Den sjekker om størrelsen på bufferen som brukeredelen av receive er stor nok. Om den er for liten settes et flagg som sier ifra til brukerprosessen for å advare den. Det siste funksjonen gjør er å sjekke om meldingen var sent som en synkronisert melding eller asynkron. Om meldingen ble sent som synkron betyr det at den sendende prosessen venter på å få klarsignal på å kjøre videre. Funksjonen setter sendende prosess' tilstand til READY og dytter den på køen til planleggeren.

```
1  #define QUEUE_EMPTY          (1 << 0)
2  #define BUF_TOO_SMALL        (1 << 1)
3
4  scheduling_type_t system_receive(ipc_msg_t *recv_msg, size_t buf_size, int* flags){
5      int cpy_bytes;
6      PCB_t* my_pcb = pcb_get( get_current_running_process() );
7      ipc_msg_t* popped_msg = ipc_dequeue(get_current_running_process());
8      if ( popped_msg != NULL){
9          if( popped_msg->payload_size > buf_size){
10             *flags |= BUF_TOO_SMALL;
11             cpy_bytes = buf_size;
12         }else{
13             cpy_bytes = popped_msg->payload_size;
14         }
15         memcpy(       (void*)recv_msg,
16                       (void*)popped_msg,
17                       sizeof(ipc_msg_t) + cpy_bytes);
18         if( popped_msg->flags & WAITING_SEND ){
19             pcb_get(recv_msg->sender)->state=READY;
20             scheduler_enqueue(recv_msg->sender);
21         }
22         free(popped_msg);
23     }else{
24         my_pcb->state = BLOCKED_RECEIVING;
25         *flags |= QUEUE_EMPTY;
26     }
27     return RESCHEDULE;
28 }
```

10 Drivere

En helt nødvendig del av operativsystemet er drivere. Det må defineres et driver-grensesnitt for prosesser slik at de kan registrere seg som drivere. En drivere er en prosess som sier at de vil ha ansvar for en enhet. Den største forskjellen mellom drivere og vanlige prosesser er at drivere vil få en melding fra kjernen når deres enhet gir ett avbrudd som driveren må behandle. Det finnes også drivere som ikke har ansvar for noen enhet, men kan være nyttig for å implementere protokoller og lignende som ikke har egne enheter.

Drivergrensesnittet er følgende funksjoner:

Driver_register Registrere prosessen som en ny driver.

Driver_get Finne driveren til en enhet.

Driver_remove Fjerner en driver.

10.1 Holde styr på drivere

Kjernen lagrer drivere in en lenket liste, det hver node inneholder drivernavnet, prosessidentiteten og en peker til neste node. Dette er implementert på linje 1-7. Hver node har et sett med enheter som trenger en driver. Kjernen holder styr på drivere ved å lagre en driver for hver enhet. Denne lista vil initialiseres til å inneholde NULL verdier. Når prosesser begynner å registrere seg som drivere blir lista fylt. Dette er implementert på linje 9-22.

```
1  typedef struct driver{
2      char name[DRIVER_NAME_SIZE];
3      process_id_t id;
4      struct driver* next;
5  }driver_t;
6
7  driver_t *os_drivers;
8
9  struct driver_irqs{
10     process_id_t timer;
11     process_id_t gpio;
12     process_id_t uart;
13     // add more as needed!
14 };
15
```

```
16 static struct driver_irqs irq_drivers;
17
18 void drivers_init(void){
19     irq_drivers.timer = NULL_ID;
20     irq_drivers.gpio = NULL_ID;
21     irq_drivers.uart = NULL_ID;
22 }
```

Som sett i avbruddshåndtern interrupt_vector_c vil alle avbrudd som genereres av en enhet føre til at kjernen sender en melding til driveren som styrer enheten. Funksjonen driver_irq_get finner driveren som skal drive enheten ved å se på driverlista.

```
1  process_id_t driver_get(char* name){
2      driver_t *node = os_drivers;
3      while(node){
4          if( strncmp(node->name, name, DRIVER_NAME_SIZE) == 0){
5              // has found driver
6              return node->id;
7          }
8          node = node->next;
9      }
10     // did not find driver
11     return NULL_ID;
12 }
13
14 uint32_t interrupt_vector_c(void){
15 //  find irq source
16     irq_controller_t *irq_flags = irq_controller_get();
17     //  timer
18     if( irq_flags->IRQ_basic_pending & ARM_TIMER_IRQ ){
19         ipc_kernel_send(NULL, 0, driver_irq_get(DRIVER_TIMER));
20         return time_handler();
21     }
22     // uart
23     if ( irq_flags->IRQ_pending_2 & UART_IRQ){
24         ipc_kernel_send(NULL, 0, driver_irq_get(DRIVER_UART));
25         return uart_handler();
26     }
27     return 0;
28 }
```

10.2 Registrere driver

Drivere kan registrere seg med systemkallet driver_register. Et av argumentet er en char-string som inneholder navnet på enheten driveren skal ha. Driver_register lager en node og legger denne i den lenkede listen. Videre sjekker den om drivernavnet matcher navnet til en enhet. Om den matcher skrives denne nye driveren som driveren til enheten. Da vil denne driveren få en melding fra kjernen

hver gang enheten genererer et avbrudd.

```
1     //  process side
2  int driver_register(char *name){
3      int errors;
4      _SYSTEM_CALL(DRIVER_REGISTER, (void*)name, (void*)&errors, NULL);
5      return errors;
6  }
7
8      //  kernel side
9  scheduling_type_t driver_register(process_id_t id, char* name, int* errors){
10     // allocate space
11     driver_t *node = driver_create(id, name);
12     if( node == NULL){
13         uart_puts("driver_add: failed to initialize node\r\n");
14         *errors = 1;
15         return NO_RESCHEDULE;
16     }
17     node->next = os_drivers;
18     os_drivers = node;
19
20     if( strcmp( node->name, "uart" ) == 0){
21         irq_drivers.uart = id;
22     }else if(strcmp( node->name, "timer") == 0){
23         irq_drivers.timer = id;
24     }else if(strcmp( node->name, "gpio") == 0){
25         irq_drivers.gpio = id;
26     }
27     *errors = 0;
28     return NO_RESCHEDULE;
29 }
```

10.3 Slette en driver

En driver slettes når prosessen blir drept. Som sett i 8.5 vil kill funksjonen kalle funksjonen driver_remove. Først itererer den seg gjennom den linkede listen, og sletter noden med riktig identitet. Deretter ser den om driveren har ansvar for en enhet. Om den har ansvar sletter den driveren ved å skrive NULL_ID over.

```
1  int driver_remove(process_id_t id){
2      driver_t **pp = &os_drivers;
3      driver_t *del = NULL;
4      while(*pp){
5          if( pcb_id_compare( (*pp)->id, id) == 1){
6              del = *pp;
7              *pp = (*pp)->next;
8              free(del);
9              return 1;
10         }
```

```
11          pp = &(*pp)->next;
12      }
13      if( pcb_id_compare(irq_drivers.timer, id){
14          irq_drivers.timer = NULL_ID;
15      }
16      ....
17      return 0;
18  }
```

11 Laste filer fra SD-kortet

Å laste opp filer fra SD-kortet til minnet er en svært viktig funksjonalitet for kjernen. Når systemet starter er det bare kjernen som ligger i minnet, uten noen drivere rundt. Inkludert en SD-kort driver. For å begynne å laste inn prosesser må dermed kjernen ha en intern funksjonalitet for å laste filer fra SD-kortet. Det er to lag som KybOS må klare å bruke. Det første er å laste opp data fra bestemte adresser på SD-kortet. Dette utføres i et Embedded Memory Management Controller-lag. Det andre laget er å tolke og finne filer som er lagret på et FAT-formatert filsystem på SD-kortet. Dette gjøres i et FAT-lag.

11.1 Embedded Memory Management Controller-laget

Raspberry Pi 2 har en Embedded Memory Management Controller som har input-output registre i minnet. Disse registrene brukes til å kommunisere med SD kortet. Etter at en skriver en kommando er det nødvendig å bruke en timeout_wait funksjon som venter til riktig bit i et register blir satt, eller tidsfristen går ut. Om tidsfristen går ut kan kjernen gjøre en annen handling for å utbedre en feil. Denne funksjonen er implementert slik:

```
1 static int timeout_wait(volatile uint32_t *reg, uint32_t mask, int value, uint32_t msec){
2     time_unit_t time1 = time_get();
3     time_add_microseconds(&time1 ,msec);
4     do{
5         if ( (*reg & mask) ? value : !value){ return 0;}
6     } while( time_compare(time1, time_get()) == 1);
7     return -1;
8 }
```

Reg peker på registeret som skal testes. på line 2 finner funksjonen nåværende tid. På linje 3 så legges det til en tidsfrist. så kjører funksjonen i en loop til et bit i registeret som er i mask blir satt høy, eller tidsfristen går ut. Om tidsfristen går ut, returner med en feilverdi.

Dokumentasjon om EMMC og de fysiske protokollene mot SD-kortet som er brukt her er [8], som inneholder blant annet tilstandsdiagrammer for enkelte handlinger, og [9] inneholder blant annet listen over kommandoer og hva de gjør.

I dette delkapittelet vil det bli presentert 5 funksjoner fra EMMC-laget:

Emmc_command går igjennom hvordan funksjonen sender en commando til SD-kortet.

Emmc_init initialiserer SD-kortet og henter en del informasjon som lagres.

Emmc_data_ setter kortet i en modus som gjør det mulig å overføre data.

Emmc_emmc_data_ er funksjonen for å overføre data til eller fra SD-kortet.

sd_read Som er en wrapper funksjon rundt noen EMMC funksjoner får å øke fleksibiliteten og sikre at argumenter blir gitt riktig.

11.1.1 Sende en kommando

For å sende en kommando er det nødvendig å implementere tilstandsdiagrammene 11.1 og 11.2. Hver tilstand består i å skrive til et register, med en påfølgende timeout_wait for å sjekke at alt gikk riktig. Om kommandoen er å overføre data må også tilstandsdiagrammet 11.3 utføres for å fullføre kommandoen.

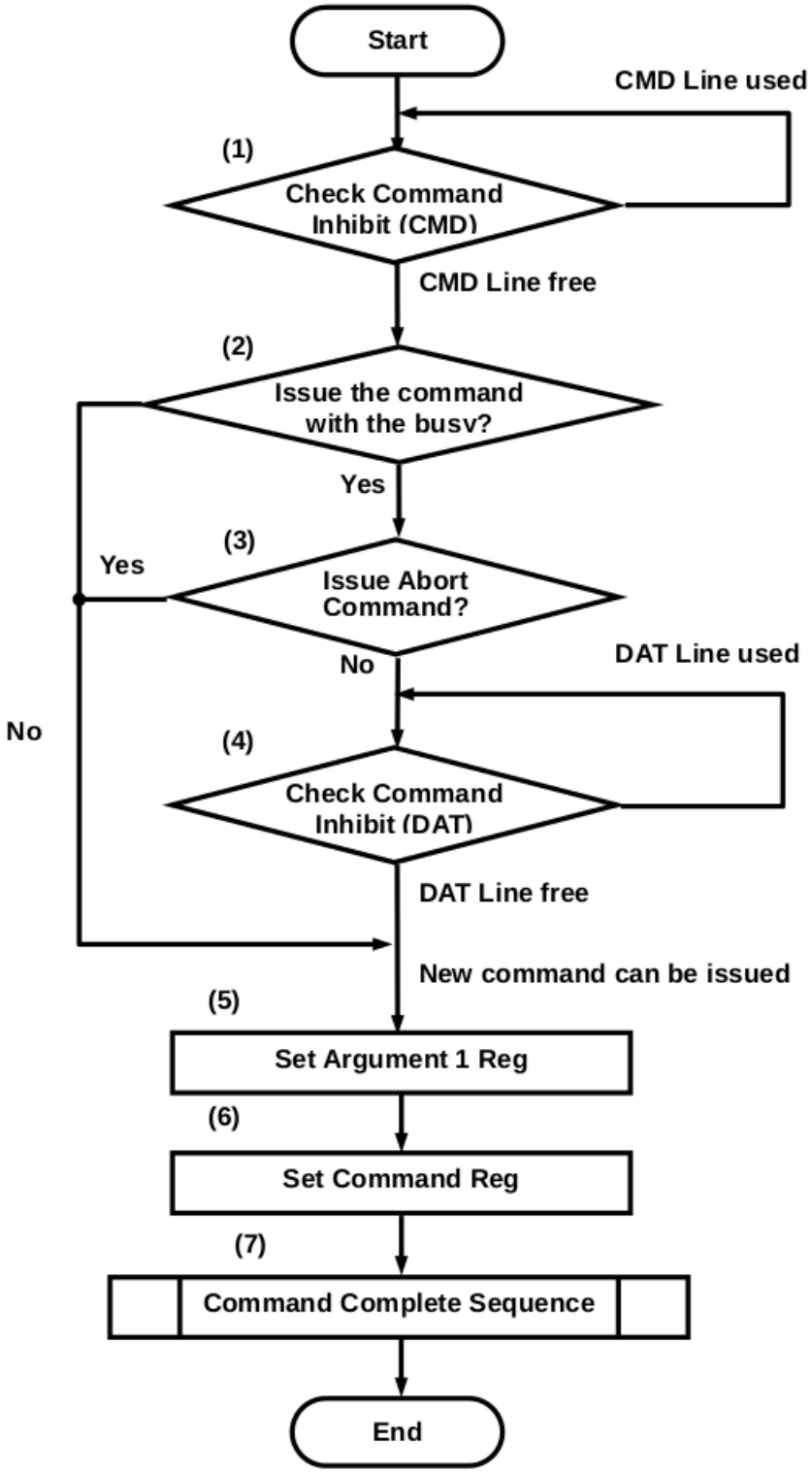
Disse tre flytdiagrammene er implementert i emmc_command_single, som sender en kommando til SD-kortet. Først venter den til det er klar til å sende en ny kommando. Dette gjøres ved å lese CMD_INHIBIT bit i STATUS registeret. Dette er linje 6-8 og svarer til tilstand 1 i figur 11.1. Deretter uføres tilstand 2, 3 og 4 på linjene 10 til 17. Dette er avhengig av hvilken kommando som sendes, og hvilken type respons som denne kommando får. Deretter skriver block_size og block_count som gjøres i tilstand 1 og 2 i figur 11.3.

Etter at disse tingene er sjekket og unnagjort er det klart for å skrive argumentet til kommandoen, altså tilstand 5 i figur 11.1 og tilstand 3 i figur 11.3. Dette gjøres på linje 29. Deretter kan kjernen sende i vei kommandoen. Når kjernen skriver til CMDIM sendes kommandoen til SD-kortet med argumentet som ble skrevet tidligere. Dette er tilstand 6 i figur 11.1 og 5 figur 11.3 og skjer på linje 32.

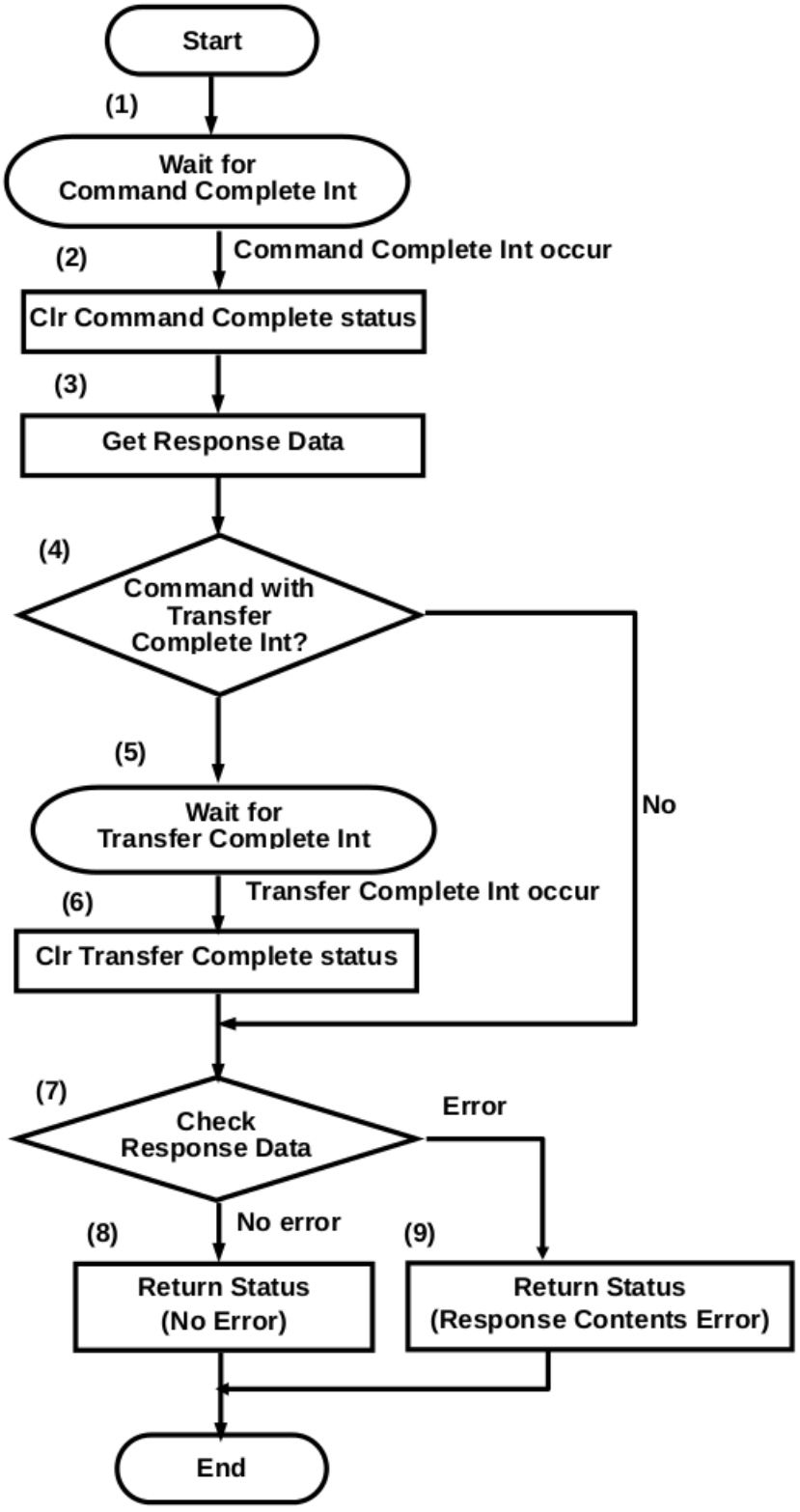
Deretter må kjernen ta imot responsen. Den må vente på kommando uført status på INTERRUPT registeret, dette gjøres på linje 36, og er tilstand 1 i figur 11.2 og 6 i 11.3. Deretter gjør den linje 40 som er å klarere kommando utført signalet, tilstand 2 og 7. Deretter tester den for feil.

Neste tilstand, tilstand 3 og 8, er å hente ut responsverdien fra RESP registrene. Dette er linje 52-63 Om kommandoen var å overføre data fra SD-kortet må denne dataen også tas imot. Denne sjekken utføres i IF setningen på linje 66. Om det er en kommando med dataoverføring, finner den ut om dataoverføringen er å skrive eller lese fra SD kortet. Den leser/skriver data fra/til en buffer i while-løkken på linje 78-103. Dette er tilstand 9 til 17 i figur 11.3.

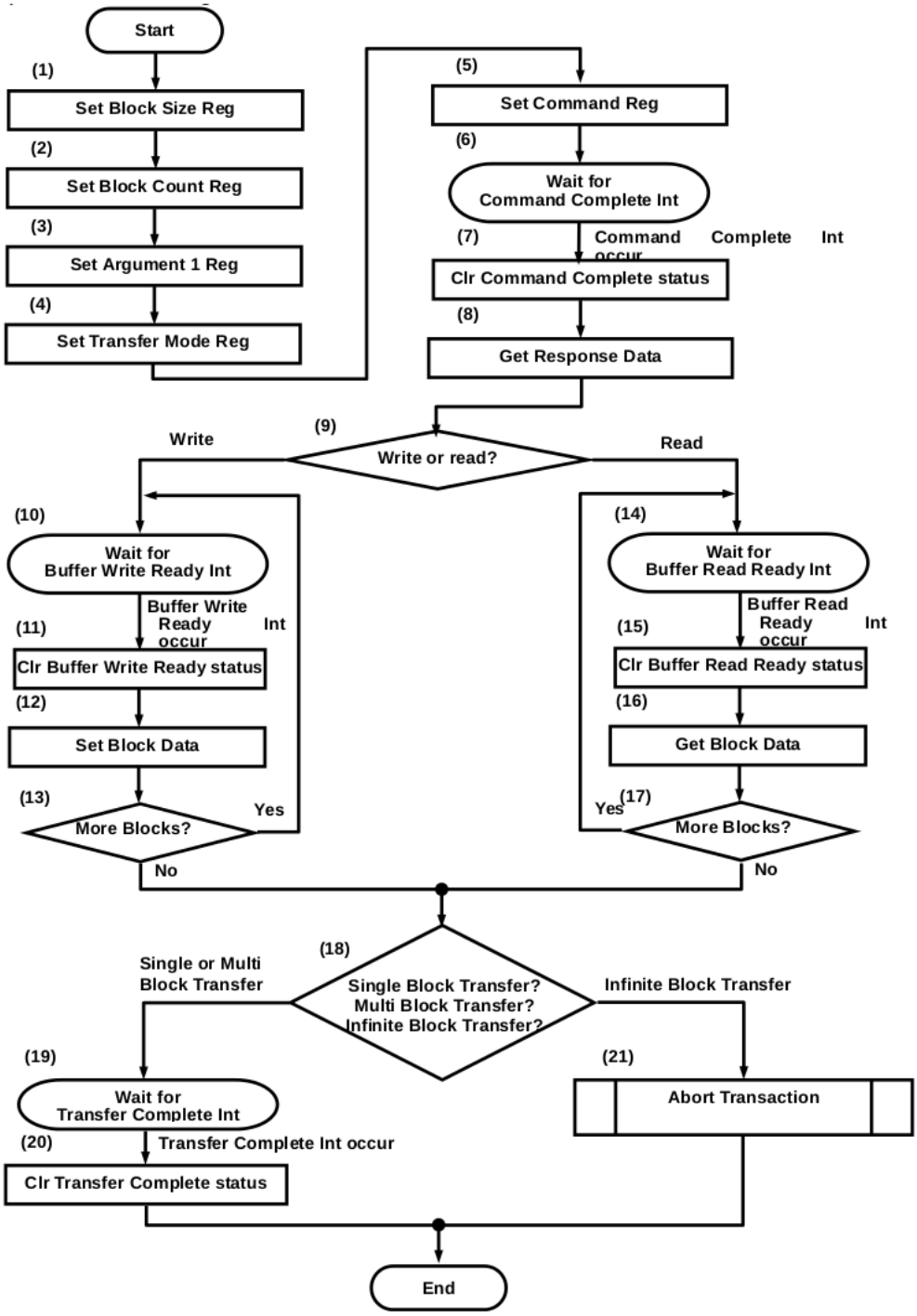
Til slutt sjekkes om kommandoen var gjennomfør riktig og at det ikke har blitt noen feil i løpet av sekvensen. Om det ikke har vært feil, returner etter å skrive at kommandoen var en suksess.



Figur 11.1: Figur 3-11 i [8], hvordan å sende en kommando del 1



Figur 11.2: Figur 3-12 i [8], "command complete sequence"



Figur 11.3: Figur 3-13 i [8], en kommando med data, uten DMA

```
1  static void emmc_command_single(          struct emmc_dev *dev,
2                                            uint32_t cmd_reg, uint32_t argument,
3                                            uint32_t timeout){
4      dev->last_cmd_reg = cmd_reg;
5      dev->last_cmd_success = 0;
6      while(emmc_get()->STATUS & CMD_INHIBIT){
7          time_delay_microseconds(1);
8      }
9      // Is the command with busy?
10     if((cmd_reg & EMMC_CMD_RSPNS_TYPE_MASK) == EMMC_CMD_RSPNS_TYPE_48B){
11         if((cmd_reg & EMMC_CMD_TYPE_MASK) != EMMC_CMD_TYPE_ABORT){
12             // Wait for the data line to be free
13             while( emmc_get()->STATUS & DAT_INHIBIT){
14                 time_delay_microseconds(1);
15             }
16         }
17     }
18
19     // Set block size and block count
20     // block size = 512 bytes, block count = 1,
21     if(dev->blocks_to_transfer > 0xffff){
22         dev->last_cmd_success = 0;
23         return;
24     }
25     uint32_t blksizecnt = dev->block_size | (dev->blocks_to_transfer << 16);
26     emmc_get()->BLKSIZECNT = blksizecnt;
27
28     // Set argument 1 reg
29     emmc_get()->ARG1 = argument;
30
31     // Set command reg
32     emmc_get()->CMDIM = cmd_reg;
33     time_delay_microseconds(2);
34
35     // Wait for command complete or error
36     timeout_wait( &(emmc_get()->INTERRUPT), (ERR | CMD_DONE), 1, timeout);
37     uint32_t interrupts = emmc_get()->INTERRUPT;
38
39     // Clear command complete status
40     emmc_get()->INTERRUPT = 0xffff0001;
41
42     // Test for errors
43     if((interrupts & (ERROR_MASK | CMD_DONE)) != CMD_DONE){
44         dev->last_error = interrupts & ERROR_MASK;
45         dev->last_interrupt = interrupts;
46         return;
47     }
48
49     time_delay_microseconds(2);
50
51     // Get response data
52     switch(cmd_reg & EMMC_CMD_RSPNS_TYPE_MASK){
```

```
53          case EMMC_CMD_RSPNS_TYPE_48:
54          case EMMC_CMD_RSPNS_TYPE_48B:
55              dev->last_r0 = emmc_get()->RESP0;
56              break;
57          case EMMC_CMD_RSPNS_TYPE_136:
58              dev->last_r0 = emmc_get()->RESP0;
59              dev->last_r1 = emmc_get()->RESP1;
60              dev->last_r2 = emmc_get()->RESP2;
61              dev->last_r3 = emmc_get()->RESP3;
62              break;
63      }
64
65      // If with data, wait for the appropriate interrupt
66      if(cmd_reg & EMMC_CMD_WITH_DATA){
67          uint32_t wr_irq;
68          int is_write = 0;
69          if(cmd_reg & EMMC_CMD_DAT_DIR_CH){
70              wr_irq = READ_RDY;
71          }else{
72              is_write = 1;
73              wr_irq = WRITE_RDY;
74          }
75
76          int cur_block = 0;
77          uint32_t *cur_buf_addr = (uint32_t *)dev->buf;
78          while(cur_block < dev->blocks_to_transfer){
79              timeout_wait( &(emmc_get()->INTERRUPT), wr_irq | ERR, 1, timeout);
80              interrupts = emmc_get()->INTERRUPT;
81              emmc_get()->INTERRUPT = ERROR_MASK | wr_irq;
82              // check for error
83              if((interrupts & (ERROR_MASK | wr_irq)) != wr_irq){
84                  dev->last_error = interrupts & ERROR_MASK;
85                  dev->last_interrupt = interrupts;
86                  return;
87              }
88
89              // Transfer the block
90              size_t cur_byte_no = 0;
91              while(cur_byte_no < dev->block_size){
92                  if(is_write){
93                                  uint32_t data = read_word((uint8_t *)cur_buf_addr, 0);
94                      emmc_get()->DATA = data;
95                                  }else{
96                                          uint32_t data = emmc_get()->DATA;
97                      write_word(data, (uint8_t *)cur_buf_addr, 0);
98                                  }
99                  cur_byte_no += 4;
100                 cur_buf_addr++;
101             }
102             cur_block++;
103         }
104     }
105
```

```
106     // Wait for transfer complete (set if read/write transfer or with busy)
107     if(((cmd_reg & EMMC_CMD_RSPNS_TYPE_MASK) == EMMC_CMD_RSPNS_TYPE_48B) ||
108        (cmd_reg & EMMC_CMD_WITH_DATA)){
109         // First check command inhibit (DAT) is not already 0
110         if( ((emmc_get()->STATUS) & DAT_INHIBIT) == 0){
111             emmc_get()->INTERRUPT = ERROR_MASK | DATA_DONE;
112         }else{
113             timeout_wait( &(emmc_get()->INTERRUPT), ERR | DATA_DONE, 1, timeout);
114             interrupts = emmc_get()->INTERRUPT;
115             emmc_get()->INTERRUPT = ERROR_MASK | DATA_DONE;
116 
117             // Handle the case where both data timeout and transfer complete
118             //  are set - transfer complete overrides data timeout: HCSS 2.2.17
119             if(((interrupts & (ERROR_MASK | DATA_DONE)) !=
120                     DATA_DONE) && ((interrupts & (ERROR_MASK | DATA_DONE))
121                     != (DTO_ERR | DATA_DONE))){
122                 dev->last_error = interrupts & ERROR_MASK;
123                 dev->last_interrupt = interrupts;
124                 return;
125             }
126             emmc_get()->INTERRUPT = ERROR_MASK | DATA_DONE;
127         }
128     }
129     // Return success
130     dev->last_cmd_success = 1;
131 }
```

Det er to typer kommandoer som SD-kortet bruker. Disse er kommandoer og applikasjons kommandoer. dokumentasjonen til disse kommandoene kan ses i [9]. Applikasjons kommandoer er en applikasjons-spesisifikk kommando, og har samme struktur som vanlige kommandoer. For å sende en applikasjons kommando sendes først en vanlig kommando (APP_CMD, nummer 55) som forteller SD-kortet at den neste kommando er en applikasjonskommando.

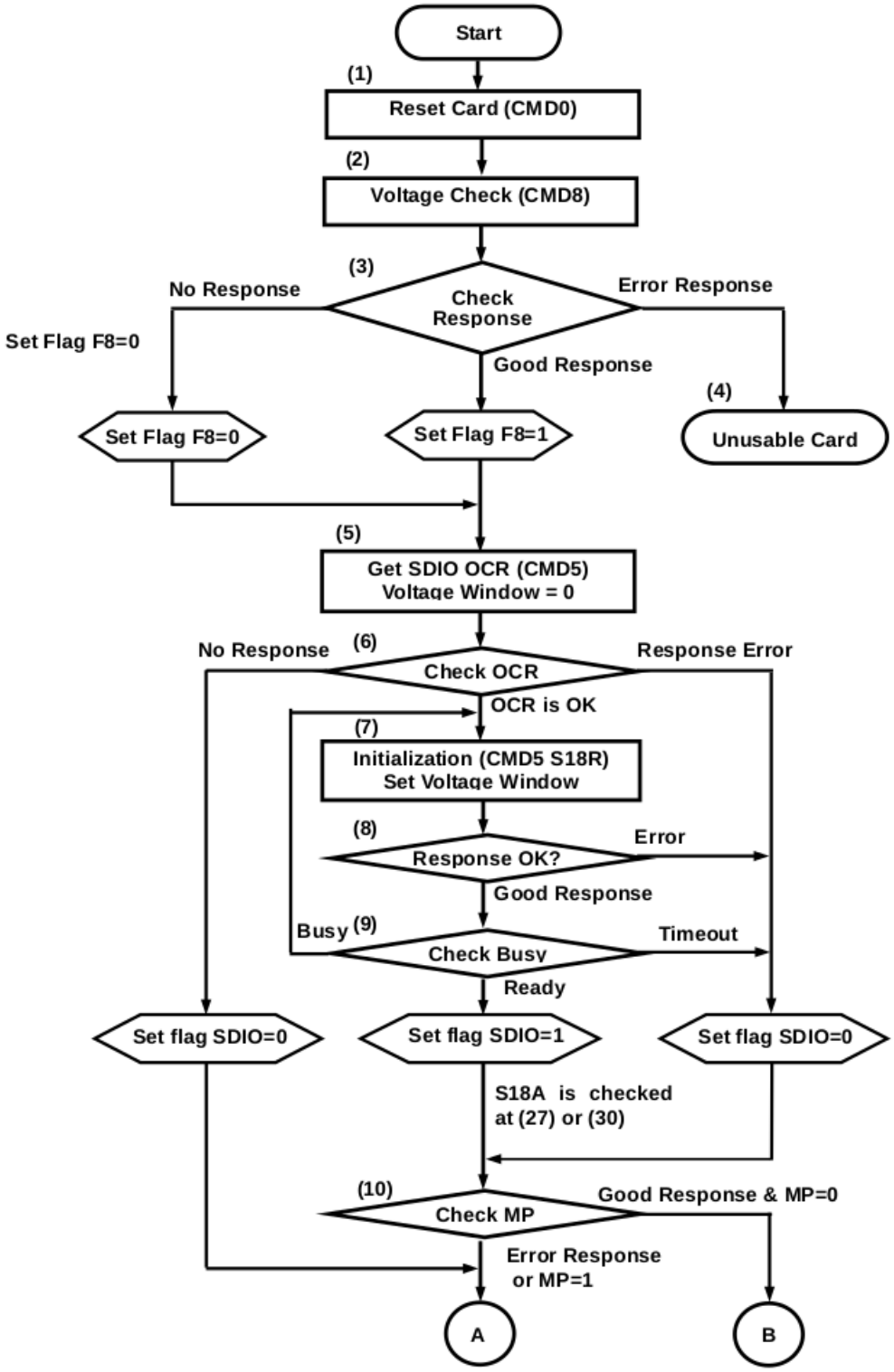
Denne splitten utføres i emmc_command. Først sjekker den om commandoen er en applikasjonskommando. Om det ikke er en applikasjonskommando sendes bare kommandoen til SD-kortet ved hjelp av emmc_command_single. Dette gjøres på linje 34-41. Om det er en applikasjons kommando sender funksjonen først en APP_CMD, og om den ikke gir en feil sendes applikasjons-kommandoen etterpå. Disse applikasjons kommandoene varier ut ifra hva slags SD-kort som finnes, men det er noen som er standart blant alle SD-kort.

```
1  static int emmc_command(        struct emmc_dev *dev,
2                                  uint32_t command,
3                                  uint32_t argument,
4                                  uint32_t timeout){
5      // First, handle any pending interrupts
6      emmc_handle_interrupts(dev);
7      // Stop the command issue if it was the card remove interrupt that was
8      //  handled
9      if(dev->card_removal){
10         dev->last_cmd_success = 0;
11         return -1;
```

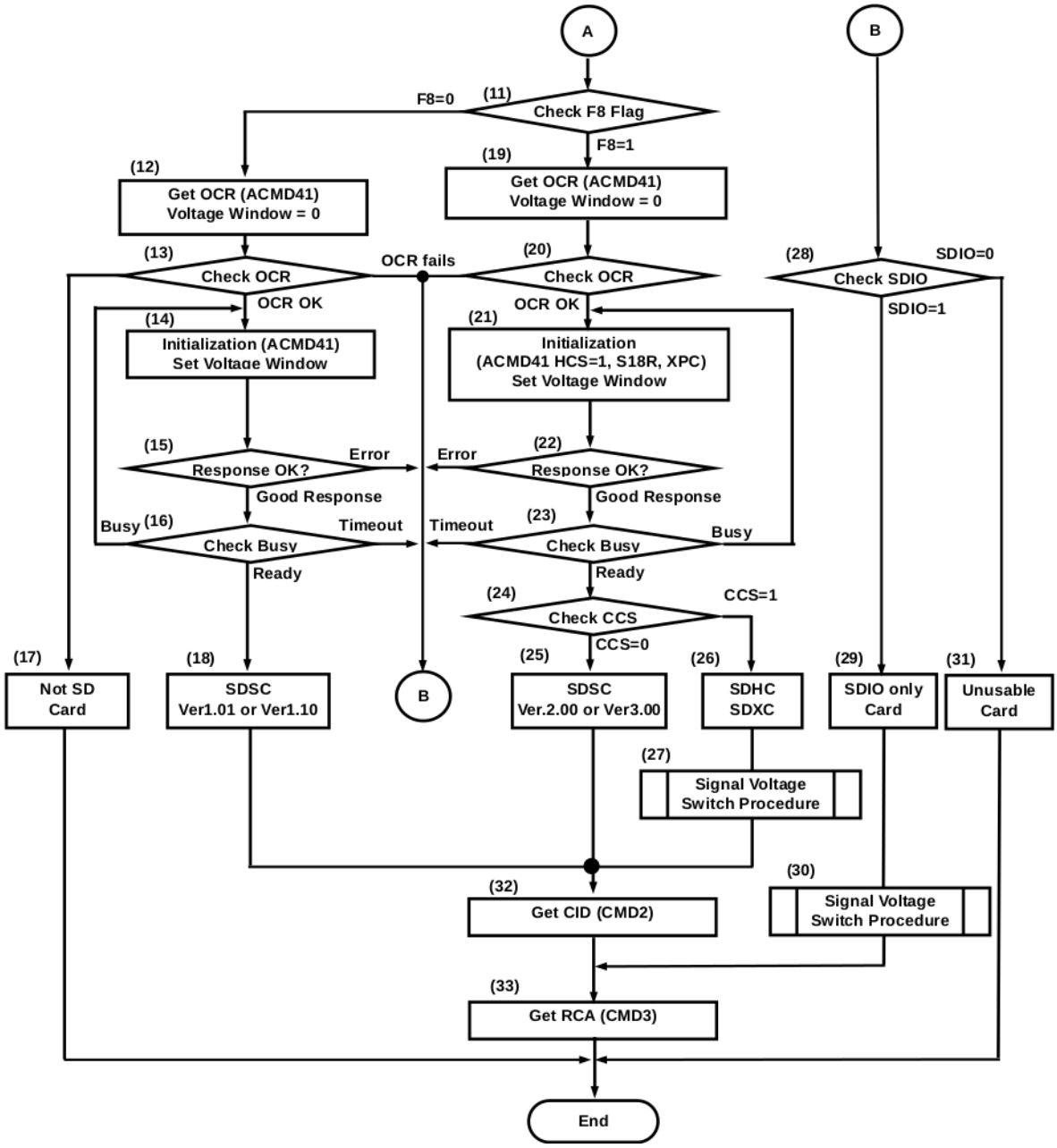
```
12      }
13
14      // Now run the appropriate commands by calling emmc_issue_command_int()
15      // mask out IS_APP_CMD
16      if(command & IS_APP_CMD){
17          command &= 0xff;
18
19          if(emmc_acommands[command] == EMMC_CMD_RESERVED(0)){
20              dev->last_cmd_success = 0;
21              return -1;
22          }
23          dev->last_cmd = APP_CMD;
24
25          uint32_t rca = 0;
26          if(dev->card_rca){
27              rca = dev->card_rca << 16;
28          }
29          emmc_command_single(dev, emmc_commands[APP_CMD], rca, timeout);
30          if(dev->last_cmd_success){
31              dev->last_cmd = command | IS_APP_CMD;
32              emmc_command_single(dev, emmc_acommands[command], argument, timeout);
33          }
34      }else{
35          if(emmc_commands[command] == EMMC_CMD_RESERVED(0)){
36              dev->last_cmd_success = 0;
37              return -1;
38          }
39          dev->last_cmd = command;
40          emmc_command_single(dev, emmc_commands[command], argument, timeout);
41      }
42      return 0;
43  }
```

11.1.2 Initialisere kortet

Å initialisere SD-kortet er en ganske lang prosess. Flytdiagrammet for å initialisere SD-kortet finnes i [8].



Figur 11.4: Del en av figur 3-9 i [8], initialiseringen av SD kortet



Figur 11.5: Del to av figur 3-9 i [8], initialiseringen av SD kortet

Disse tilstandsdiagrammene er implementert i emmc_card_init. Det første den gjør er å skru av og på strømmen til EMMC-denheten. Dette sikrer at at den er i startup tilstanden. Dette gjøres til linje Deretter gjør den det samme med SD-kortet. Deretter initialiserer den klokker for enheten. på linje 87-93 initialiserer funksjonen en struct som skal inneholde en del informasjon som kjernen må huske på.

Tilstand 1 i 11.4 er å sende kommando 0, GO_IDLE_STATE. Dette gjør at SD-kortet går til IDLE tilstanden, og er implementert på linje 99. Deretter sjekkes det for feil. Tilstand 2, å sende kommando 8, SEND_IF_COND, som sender med strøminformasjon. Tilbake som svar får funksjonen en respons, som sier om kortet er brukbart eller ikke. Om det ikke er mulig å bruke kortet returnerer funksjonen med en feilverdi. Dette er implementert på linje 108 til 130.

Om kortet er brukbart vil funksjonen gå til tilstand 5, GET_OCR. Dette gjøres ved å sende kommando 5 (IO_SET_OP_COND). Dette gjøres på linje 134-146. Det neste tilstanden er å sende applikasjonskommando 41. Fordi det er gjort noen forenklinger ha det blitt antatt at F8 alltid vil være lik en (altså at man får en god respons på kommando 8 i tilstand 2). Dette gjør at man nå befinner seg i tilstand 19. Initialiseringsløkken ved tilstand 21 til 23 er implementert på linje 156 til 187.

Deretter sjekker funksjonen om kortet støtter 1,8 volt. I så fall må tilstandsdiagrammet 11.6 implementeres. Denne er implementert i en if setning fra linje 195 til 251. Først sendes kommando 11 (Voltage_SWITCH). Om responsen er god går den videre til neste steg, som er å skru av klokke signalet, og sjekke DAT bit-ene i STATUS registeret. Etter en liten ventetid skrus klokkesignalet på igjen. Til slutt sjekkes DAT bit-ene igjen for om de inneholder en vellykket verdi. Om denne sekvensen var vellykket har man nå byttet spenning.

Den neste tilstanden er nummer 32 hvor det sendes kommando nummer 2, ALL_GET_CID. Denne får SD-kortet til å sende sitt Card Identification Number. Dette gjøres på linje 254. CID nummeret blir deretter lagret for å brukes senere. Den siste tilstanden i tilstandsdiagrammet er nummer 33. Her sendes kommando nummer 3 (SEND_RELATIVE_ADDRESS). Denne verdien lagres også. De siste cirka 100 linjene er stort sett å innhente informasjon som trengs fra SD-kortet og å sjekke for feil. Kommandoene som sendes er SEND_SCR, for å få kortets Card Configuration Register, SET_BLOCKLEN for å være sikker på at én blokk har en lengde på 512, og om buss bredden er 1, justeres den til 4. Dette øker farten på å overføre data. Dette er en enkel sekvens som er implementert slik at det oppfyller tilstandsdiagrammet 11.7, på linje 371 til 393. Når funksjonen er ferdig er SD-kortet klart til å skrive og lese data.

```
1  static int emmc_card_init(struct emmc_dev **dev){
2      if(bcm_emmc_power_cycle() != 0){
3          uart_puts(''EMMC init: failed power cycle\r\n'');
4          return -1;
5      }
6
7      // read the sd controller version
8      uint32_t ver = emmc_get()->SLOTISR_VER;
9      uint32_t emmcversion = (ver >> 16) & 0xff;
10     sd_version = emmcversion;
11
12     if(sd_version < 2){
```

```
13          uart_puts(''EMMC init: sd version not supported\r\n'');
14          return -1;
15      }
16
17      // Reset the controller
18      uint32_t control1 = emmc_get()->CONTROL1;
19      control1 |= SRST_HC;
20      // Disable clock
21      control1 &= ~CLK_EN;
22      control1 &= ~CLK_INTLEN;
23      emmc_get()->CONTROL1 = control1;
24      if (timeout_wait( &(emmc_get()->CONTROL1), SRST_DATA | SRST_CMD | SRST_HC, 0, 10) < 0){
25          uart_puts(''EMMC init: clock disable timout wait error\r\n'');
26          return -1;
27      }
28      if((emmc_get()->CONTROL1 & (SRST_DATA | SRST_CMD | SRST_HC)) != 0){
29          uart_puts(''EMMC init: clock disable bits still high\r\n'');
30          return -1;
31      }
32
33          // Check for a valid card
34      timeout_wait( &(emmc_get()->STATUS), VALID_CARD, 1, 500);
35      uint32_t status_reg = emmc_get()->STATUS;
36      if((status_reg & VALID_CARD) == 0){
37          uart_puts(''EMMC init: invalid card\r\n'');
38          return -1;
39      }
40      // Clear control2
41      emmc_get()->CONTROL2 = 0;
42
43      // Get the base clock rate
44      uint32_t base_clock = emmc_get_base_clock();
45      if(base_clock == 0){
46          uart_puts(''EMMC init: cant get emmc base clock. assuming 100 000 000\r\n'');
47          base_clock = 100000000;
48      }
49      // Set clock rate to something slow
50      control1 = emmc_get()->CONTROL1;
51      control1 |= CLK_INTLEN;     // enable clock
52
53      // Set to identification frequency (400 kHz)
54      uint32_t f_id = emmc_get_clock_divider(base_clock, EMMC_CLOCK_ID);
55      if(f_id == EMMC_GET_CLOCK_DIVIDER_FAIL){
56          uart_puts(''EMMC init: get clock divider failed\r\n'');
57          return -1;
58      }
59      control1 |= f_id;
60
61      control1 |= DATA_TOUNIT(7);          // data timeout = TMCLK * 2^10
62      emmc_get()->CONTROL1 = control1;
63      timeout_wait( &(emmc_get()->CONTROL1), CLK_STABLE, 1, 0x100);
64      if((emmc_get()->CONTROL1 & CLK_STABLE) == 0){
65          uart_puts(''EMMC init: clock adjustment timeout wait failed\r\n'');
```

```
66          return -1;
67      }
68
69      // Enable the EMMC clock
70      time_delay_microseconds(2);
71      control1 = emmc_get()->CONTROL1;
72      control1 |= CLK_EN;
73      emmc_get()->CONTROL1 = control1;
74      time_delay_microseconds(2);
75
76      // Mask off sending interrupts to the ARM
77      emmc_get()->IRPT_EN = 0;
78      // Reset interrupts
79      emmc_get()->INTERRUPT = 0xffffffff;
80      // Have all interrupts sent to the INTERRUPT register
81      uint32_t irpt_mask = 0xffffffff & (~CARD);
82      irpt_mask |= CARD;
83      emmc_get()->IRPT_MASK = irpt_mask;
84
85      time_delay_microseconds(2);
86
87      // Prepare the device
88      struct emmc_dev *ret;
89      if(*dev == NULL){
90          ret = (struct emmc_dev *)malloc(sizeof(struct emmc_dev));
91      }else{
92          ret = *dev;
93      }
94
95      memset(ret, 0, sizeof(struct emmc_dev));
96      ret->base_clock = base_clock;
97
98      // Send CMD0 to the card (reset to idle state)
99      emmc_command(ret, GO_IDLE_STATE, 0, 500);
100     if(FAIL(ret)){
101         uart_puts("EMMC init: GO_IDLE_STATE failed\r\n");
102         return -1;
103     }
104
105     // Send CMD8 to the card
106     // Voltage supplied = 0x1 = 2.7-3.6V (standard)
107     // Check pattern = 10101010b (as per PLSS 4.3.13) = 0xAA
108     emmc_command(ret, SEND_IF_COND, 0x1aa, 500);
109     int v2 = 0;
110     if(TIMEOUT(ret)){   v2 = 0; }
111     else if(CMD_TIMEOUT(ret)){
112         if(emmc_reset_cmd() == -1){
113             uart_puts("EMMC init: SEND_IF_COND reset cmd failed\r\n");
114             return -1;
115         }
116         emmc_get()->INTERRUPT = CTO_ERR;
117         v2 = 0;
118     }
```

```
119     else if(FAIL(ret)){
120         uart_puts(''EMMC init; SEND_IF_COND failed!\r\n'');
121         return -1;
122     }else{
123 
124         if((ret->last_r0 & 0xfff) != 0x1aa){
125             uart_puts(''EMMC: not usable card\n'');
126             return -1;
127         }else{
128             v2 = 1;
129         }
130     }
131 
132     // Here we are supposed to check the response to CMD5 (HCSS 3.6)
133     // It only returns if the card is a EMMCIO card
134     emmc_command(ret, IO_SET_OP_COND, 0, 10);
135     if(!TIMEOUT(ret)){
136         if(CMD_TIMEOUT(ret)){
137             if(emmc_reset_cmd() == -1){
138                 uart_puts(''EMMC init: Set op cond reset cmd error\r\n'');
139                 return -1;
140             }
141             emmc_get()->INTERRUPT = CTO_ERR;
142         }else{
143             uart_puts(''EMMC init: Set op cond error!\r\n'');
144             return -1;
145         }
146     }
147 
148     // Call an inquiry ACMD41 (voltage window = 0) to get the OCR
149     emmc_command(ret, ACMD(41), 0, 500);
150     if(FAIL(ret)){
151         uart_puts(''EMMC init: Get OCR failed\r\n'');
152         return -1;
153     }
154     // Call initialization ACMD41
155     int card_is_busy = 1;
156     while(card_is_busy){
157         uint32_t v2_flags = 0;
158         if(v2){
159             // Set EMMCHC support
160             v2_flags |= (1 << 30);
161 
162             // Set 1.8v support
163             if(!ret->failed_voltage_switch){
164                 v2_flags |= (1 << 24);
165             }
166 
167             // Enable EMMCXC maximum performance
168             v2_flags |= (1 << 28);
169         }
170         emmc_command(ret, ACMD(41), 0x00ff8000 | v2_flags, 500);
171         if(FAIL(ret)){
```

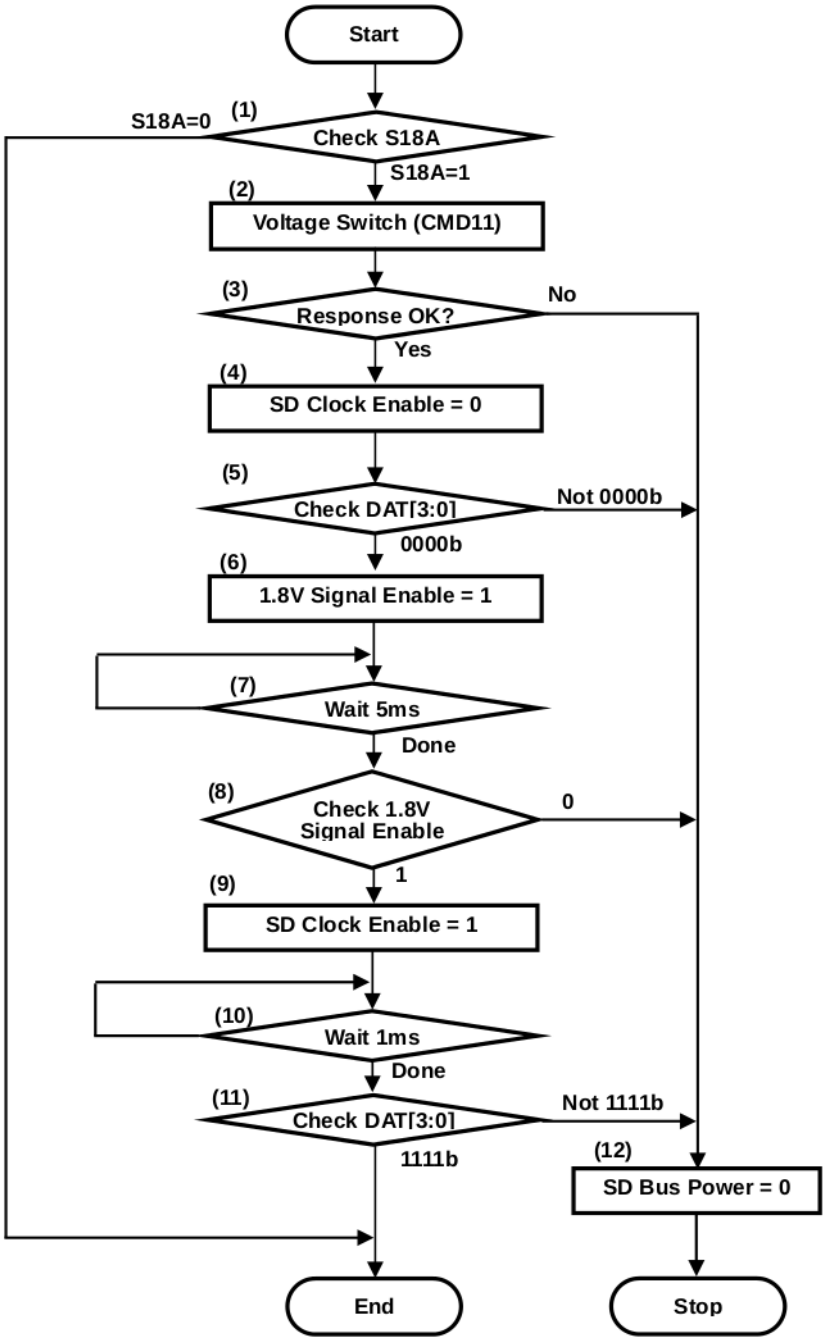
```
172             uart_puts(''EMMC: error issuing ACMD41\n'');
173             return -1;
174         }
175         if((ret->last_r0 >> 31) & 0x1){
176             // Initialization is complete
177             ret->card_ocr = (ret->last_r0 >> 8) & 0xffff;
178             ret->card_supports_sdhc = (ret->last_r0 >> 30) & 0x1;
179             if(!ret->failed_voltage_switch){
180                 ret->card_supports_18v = (ret->last_r0 >> 24) & 0x1;
181             }
182             card_is_busy = 0;
183         }else{
184             // Card is still busy
185             time_delay_microseconds(5);
186         }
187     }
188 
189     emmc_switch_clock_rate(base_clock, EMMC_CLOCK_NORMAL);
190 
191     // A small wait before the voltage switch
192     time_delay_microseconds(5);
193 
194     // Switch to 1.8V mode if possible
195     if(ret->card_supports_18v){
196         // As per HCSS 3.6.1
197 
198         // Send VOLTAGE_SWITCH
199         emmc_command(ret, VOLTAGE_SWITCH, 0, 500);
200         if(FAIL(ret)){
201             ret->failed_voltage_switch = 1;
202             emmc_power_off();
203             uart_puts(''EMMC: voltage switch error\r\n'');
204             return emmc_card_init(&ret);
205         }
206         // Disable EMMC clock
207         control1 = emmc_get()->CONTROL1;
208         control1 &= ~CLK_EN;
209         emmc_get()->CONTROL1 = control1;
210 
211         // Check DAT[3:0]
212         status_reg = emmc_get()->STATUS;
213         uint32_t dat30 = (status_reg >> 20) & 0xf;
214         if(dat30 != 0){
215             ret->failed_voltage_switch = 1;
216             emmc_power_off();
217             uart_puts(''EMMC: dat30 error\r\n'');
218             return emmc_card_init(&ret);
219         }
220         // Set 1.8V signal enable to 1
221         uint32_t control0 = emmc_get()->CONTROL0;
222         control0 |= ENABLE_1_8V;
223         emmc_get()->CONTROL0 = control0;
224         // Wait 5 ms
```

```
225         time_delay_microseconds(5);
226
227         // Check the 1.8V signal enable is set
228         control0 = emmc_get()->CONTROL0;
229         if((control0 & ENABLE_1_8V) == 0){
230             ret->failed_voltage_switch = 1;
231             emmc_power_off();
232             return emmc_card_init(&ret);
233         }
234
235         // Re-enable the EMMC clock
236         control1 = emmc_get()->CONTROL1;
237         control1 |= CLK_EN;
238         emmc_get()->CONTROL1 = control1;
239
240         // Wait 1 ms
241         time_delay_microseconds(1);
242
243         // Check DAT[3:0]
244         status_reg = emmc_get()->STATUS;
245         dat30 = (status_reg >> 20) & 0xf;
246         if(dat30 != 0xf){
247                 ret->failed_voltage_switch = 1;
248                 emmc_power_off();
249                 return emmc_card_init(&ret);
250            }
251     }
252
253     // Send CMD2 to get the cards CID
254     emmc_command(ret, ALL_SEND_CID, 0, 500);
255     if(FAIL(ret)){
256         uart_puts(''EMMC: error sending ALL_SEND_CID\n'');
257         return -1;
258     }
259     uint32_t card_cid_0 = ret->last_r0;
260     uint32_t card_cid_1 = ret->last_r1;
261     uint32_t card_cid_2 = ret->last_r2;
262     uint32_t card_cid_3 = ret->last_r3;
263
264     uint32_t *dev_id = (uint32_t *)malloc(4 * sizeof(uint32_t))
265
266     dev_id[0] = card_cid_0;
267     dev_id[1] = card_cid_1;
268     dev_id[2] = card_cid_2;
269     dev_id[3] = card_cid_3;
270
271     // Send CMD3 to enter the data state
272     emmc_command(ret, SEND_RELATIVE_ADDR, 0, 500000);
273     if(FAIL(ret)){
274         free(ret);
275         uart_puts(''EMMC init: enter data state error\r\n'');
276         return -1;
277     }
```

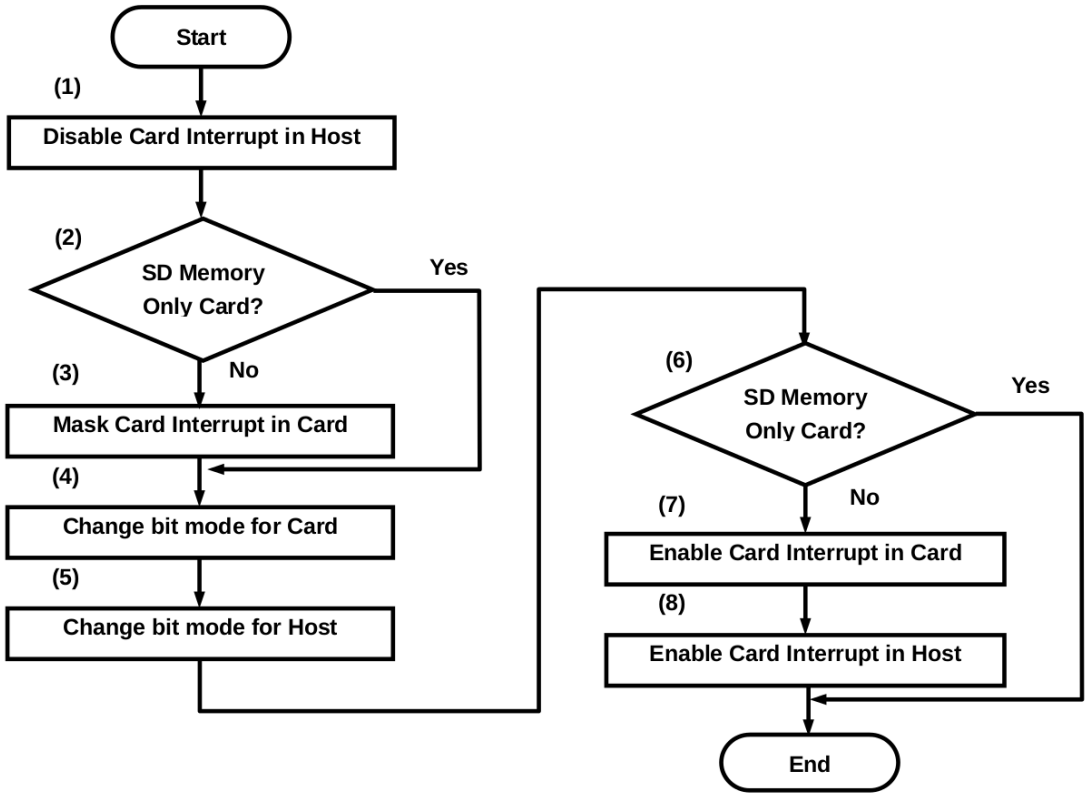
```
278     uint32_t cmd3_resp = ret->last_r0;
279
280     ret->card_rca = (cmd3_resp >> 16) & 0xffff;
281     uint32_t crc_error = (cmd3_resp >> 15) & 0x1;
282     uint32_t illegal_cmd = (cmd3_resp >> 14) & 0x1;
283     uint32_t error = (cmd3_resp >> 13) & 0x1;
284     uint32_t status = (cmd3_resp >> 9) & 0xf;
285     uint32_t ready = (cmd3_resp >> 8) & 0x1;
286
287     if(crc_error){
288         free(ret);
289         uart_puts(''EMMC init: CRC error\r\n'');
290             return -1;
291     }
292     if(illegal_cmd){
293         free(ret);
294         uart_puts(''EMMC init: illegal command error\r\n'');
295         return -1;
296     }
297     if(error){
298         free(ret);
299         uart_puts(''EMMC init: error\r\n'');
300         return -1;
301     }
302     if(!ready){
303         free(ret);
304         uart_puts(''EMMC init: not ready! error\r\n'');
305         return -1;
306     }
307     // Now select the card (toggles it to transfer state)
308     emmc_command(ret, SELECT_CARD, ret->card_rca << 16, 500);
309     if(FAIL(ret)){
310         free(ret);
311         uart_puts(''EMMC init: Select card error\r\n'');
312         return -1;
313     }
314     uint32_t cmd7_resp = ret->last_r0;
315     status = (cmd7_resp >> 9) & 0xf;
316
317     if((status != 3) && (status != 4)){
318         free(ret);
319         uart_puts(''EMMC init: status error\r\n'');
320         return -1;
321     }
322     // If not an EMMCHC card, ensure BLOCKLEN is 512 bytes
323     if(!ret->card_supports_sdhc){
324         emmc_command(ret, SET_BLOCKLEN, 512, 500);
325         if(FAIL(ret)){
326             uart_puts(''EMMC: error sending SET_BLOCKLEN\n\r'');
327             free(ret);
328             return -1;
329         }
330     }
```

```
331     ret->block_size = 512;
332     uint32_t controller_block_size = emmc_get()->BLKSIZECNT;
333     controller_block_size &= ~BLKSIZE(0xfff);
334     controller_block_size |= BLKSIZE(0x200);
335     emmc_get()->BLKSIZECNT = controller_block_size;
336 
337     // Get the cards SCR register
338     ret->scr = (struct emmc_scr *)malloc(sizeof(struct emmc_scr));
339     ret->buf = &ret->scr->scr[0];
340     ret->block_size = 8;
341     ret->blocks_to_transfer = 1;
342     emmc_command(ret, SEND_SCR, 0, 500);
343     ret->block_size = 512;
344     if(FAIL(ret)){
345         uart_puts(''EMMC: error sending SEND_SCR\n\r'');
346         free(ret->scr);
347         free(ret);
348         return -1;
349     }
350     // Determine card version
351     // Note that the SCR is big-endian
352     uint32_t scr0 = byte_swap(ret->scr->scr[0]);
353     ret->scr->sd_version = EMMC_VER_UNKNOWN;
354     uint32_t emmc_spec = (scr0 >> (56 - 32)) & 0xf;
355     uint32_t emmc_spec3 = (scr0 >> (47 - 32)) & 0x1;
356     uint32_t emmc_spec4 = (scr0 >> (42 - 32)) & 0x1;
357     ret->scr->sd_bus_widths = (scr0 >> (48 - 32)) & 0xf;
358     if(emmc_spec == 0)       { ret->scr->sd_version = EMMC_VER_1; }
359     else if(emmc_spec == 1) { ret->scr->sd_version = EMMC_VER_1_1; }
360     else if(emmc_spec == 2) {
361         if(emmc_spec3 == 0) {ret->scr->sd_version = EMMC_VER_2; }
362         else if(emmc_spec3 == 1){
363             if(emmc_spec4 == 0){
364                 ret->scr->sd_version = EMMC_VER_3;
365             }else if(emmc_spec4 == 1){
366                 ret->scr->sd_version = EMMC_VER_4;
367             }
368         }
369     }
370 
371     if(ret->scr->sd_bus_widths & 0x4){
372         // Set 4-bit transfer mode (ACMD6)
373         // See HCSS 3.4 for the algorithm
374 
375         // Disable card interrupt in host
376         uint32_t old_irpt_mask = emmc_get()->IRPT_MASK;
377         uint32_t new_iprt_mask = old_irpt_mask & ~CARD;
378         emmc_get()->IRPT_MASK = new_iprt_mask;
379 
380         // Send ACMD6 to change the card's bit mode
381         emmc_command(ret, SET_BUS_WIDTH, 0x2, 500);
382         if(FAIL(ret)){
383             uart_puts(''EMMC init: switch to 4-bit data mode failed\n\r'');
```

```
384             }else{
385                 // Change bit mode for Host
386                 uint32_t control0 = emmc_get()->CONTROL0;
387                 control0 |= HCTL_DWIDTH;
388                 emmc_get()->CONTROL0 = control0;
389 
390                 // Re-enable card interrupt in host
391                 emmc_get()->IRPT_MASK = old_irpt_mask;
392             }
393         }
394 
395         // Reset interrupt register
396         emmc_get()->INTERRUPT = 0xffffffff;
397         *dev = ret;
398         return 0;
399 }
```



Figur 11.6: Figur 3-9 i [8], hvordan å bytte spenning.



Figur 11.7: Figur 3-3 i [8], bytte bussbredde

11.1.3 Lese og skrive til SD-kortet

For å lese og skrive til SD-kortet brukes emmc_read og emmc_write. Disse funksjonen kaller først emmc_data_mode, som sikrer at SD-kortet er i data transfer modus. Deretter kalles emmc_data_command.

```
1  int emmc_read(uint8_t *buf, size_t buf_size, uint32_t block_no){
2          // Check the status of the card
3      if(emmc_data_mode(device) != 0){      return -1; }
4      if(emmc_do_data_command(device, 0, buf, buf_size, block_no) < 0){ return -1;}
5      return buf_size;
6  }
7
8  int emmc_write(uint8_t *buf, size_t buf_size, uint32_t block_no){
9          // Check the status of the card
10     if(emmc_data_mode(device) != 0){ return -1; }
11     if(emmc_data_command(device, 1, buf, buf_size, block_no) < 0){ return -1;}
12     return buf_size;
13 }
```

Emmc_data_mode sender først kommandoen SEND_STATUS for at SD-kortet skal sende innholdet i status registeret. Deretter leser den nåværende tilstand. Deretter følger det et sett med if setninger som sikrer at tilstanden når funksjonen avsluttes er å sende data.

```
1  static int emmc_data_mode(struct emmc_dev *edev){
2      if(edev->card_rca == 0){
3          // Try again to initialise the card
4          int ret = emmc_card_init(&edev);
5          if(ret != 0){    return ret;}
6      }
7
8      emmc_command(edev, SEND_STATUS, edev->card_rca << 16, 500);
9      if(FAIL(edev)){
10         uart_puts(''EMMC: ensure_data_mode() error sending CMD13\n\r'');
11         edev->card_rca = 0;
12         return -1;
13     }
14
15     uint32_t status = edev->last_r0;
16     uint32_t cur_state = CURRENT_STATE(status);
17     if(cur_state == CARD_IDENT){
18         // Currently in the stand-by state - select it
19         emmc_command(edev, SELECT_CARD, edev->card_rca << 16, 500);
20         if(FAIL(edev)){
21             uart_puts(''EMMC: ensure_data_mode() no response from CMD17\r\n'');
22             edev->card_rca = 0;
23             return -1;
24         }
25     }else if(cur_state == CARD_DATA){
26         // In the data transfer state - cancel the transmission
27         emmc_command(edev, STOP_TRANSMISSION, 0, 500);
28         if(FAIL(edev)){
29             edev->card_rca = 0;
```

```
30              return -1;
31          }
32
33          // Reset the data circuit
34          emmc_reset_dat();
35      }else if(cur_state != CARD_TRAN){
36          // Not in the transfer state - re-initialise
37          int ret = emmc_card_init(&edev);
38          if(ret != 0){   return ret; }
39      }
40
41      // Check again that we're now in the correct mode
42      if(cur_state != CARD_TRAN){
43          emmc_command(edev, SEND_STATUS, edev->card_rca << 16, 500000);
44          if(FAIL(edev)){
45              edev->card_rca = 0;
46              return -1;
47          }
48          status = edev->last_r0;
49          cur_state = (status >> 9) & 0xf;
50
51          if(cur_state != CARD_TRAN){
52              edev->card_rca = 0;
53              return -1;
54          }
55      }
56      return 0;
57  }
```

I emmc_data_command funksjonen gir gir bruker en parameter for å si om det er skrive eller lese, en buffer, størrelsen på bufferen og blokk nummeret som man ønsker å skrive eller lese fra. Først finner funksjonen ut om det er et Secure Digital High Capacity (SDHC) kort. Om det er et slikt kort brukes byte-adresser istedenfor blokk-adresser. Så sjekkes det om bufferen er stor nok. Deretter finner funksjonen ut hvor mange blokker som skal leses/skrives. Om bufferen ikke går opp i blokkstørrelsen returnerer funksjonen en feilmelding. Deretter kalles emmc_command for å utføre selve lesingen/skrivingen.

```
1   static int emmc_data_command(    struct emmc_dev *edev,
2                                    int is_write, uint8_t *buf,
3                                    size_t buf_size,
4                                    uint32_t block_no){
5       // PLSS table 4.20 - SDSC cards use byte addresses rather than block addresses
6       if(!edev->card_supports_sdhc)   {block_no *= 512; }
7
8       // This is as per HCSS 3.7.2.1
9       if(buf_size < edev->block_size){ return -1; }
10
11      edev->blocks_to_transfer = buf_size / edev->block_size;
12      if(buf_size % edev->block_size)     {return -1;}
13      edev->buf = buf;
14
15      int command;
```

```
16      if(is_write){
17          if(edev->blocks_to_transfer > 1){
18              command = WRITE_MULTIPLE_BLOCK;
19          }else{
20              command = WRITE_BLOCK;
21          }
22      }else{
23          if(edev->blocks_to_transfer > 1){
24              command = READ_MULTIPLE_BLOCK;
25          }else{
26              command = READ_SINGLE_BLOCK;
27          }
28      }
29
30      int retry_count = 0;
31      int max_retries = 3;
32      while(retry_count < max_retries){
33          emmc_command(edev, command, block_no, 500);
34          if(SUCCESS(edev)){
35              break;
36          }else{
37              retry_count++;
38          }
39      }
40      if(retry_count == max_retries) {
41          edev->card_rca = 0;
42          return -1;
43      }
44      return 0;
45  }
```

For å øke fleksibiliteten har emmc_read blitt wrappet i en funksjon sd_read. Denne Funksjonen sikrer emmc_blir kalt med en bufferstørelse på på en blokk størrelse, og har funksjonalitet for å laste mange blokker sekvensielt til en buffer.

```
1   int sd_read(uint8_t *buf, size_t buf_size, uint32_t block_no){
2       int buf_offset = 0;
3       uint32_t block_offset = 0;
4       do{
5           size_t to_read = buf_size;
6           if( to_read > emmc_get_dev_block_size()){
7               to_read = emmc_get_dev_block_size();
8           }
9           int ret = emmc_read(&buf[buf_offset], to_read, block_no + block_offset);
10          if(ret < 0){
11              return ret;
12          }
13          buf_offset += (int)to_read;
14          block_offset++;
15
16          if(buf_size < emmc_get_dev_block_size()){
17              buf_size = 0;
18          }else{
```

```
19              buf_size -= emmc_get_dev_block_size();
20          }
21      }while(buf_size > 0);
22      return buf_offset;
23  }
```

11.2 File Allocation Table

Et File Allocation Table(FAT) er et filsystem som ble brukt svært mye i Windows-operativsystemer. Det er et svært enkelt og robust system som gir god ytelse for små systemer. Det skalerer dårligere enn mer moderne filsystemer, noe som er grunnen til at det ikke lengre brukes på større systemer. Men på små innebygde datasystemer er det fremdeles brukt mye. Dette er filsystemet som Raspberry Pi-2 bruker.

FAT har tre minneområder i sin partisjon: BIOS Parameter Block (BPB), FAT, og til slutt ett data område for mapper og filer. BPB inneholder en del informasjon om mediumet, og hvor root-folderen er. Data regionen er delt opp i like store klynger. Størrelsen varierer fra 2KB til 32KB. En fil kan okkupere en eller flere av disse klyngene. En fil er derfor representert som en lenke av disse klyngene. Disse klyngene er ikke nødvendigvis lagret etter hverandre, men ofte fragmentert utover lagringsenheten. FAT området et kart over data-regionen. Den er et kontinuerlig området mellom BPB-en og data området. Den representerer en liste, hvor hvert indeks representerer en klynge. Den kan inneholde en av fire ting: klyngenummeret til den neste klyngen i filen, et End-Of-Chain(EOC) nummer som representerer slutten på en slik lenket liste, en spesiell verdi for en dårlig klynge, og tallet 0 for å representere en ubrukt klynge.

I denne delen av kapittelet om FAT presenteres disse funksjonene:

Fat_init initialiserer FAT modulen, leser en del informasjon fra SD-kortet og lagrer det til det skal brukes i andre funksjoner.

Fat_read_dir leser en klynge som representerer en mappe.

Fat_read leser en fil.

fat_load tar inn en path til filen, itererer seg gjennom mapper ved hjelp av fat_read_dir og finner filen som skal lastes, og kaller fat_read på den.

11.2.1 Master Boot Record

Den første logiske minneblokken på SD-kortet vil inneholde en Master Boot Record. Denne inneholder viktig informasjon for å finne partisjoner som er på sd-kortet. MBR er beskrevet i figur 11.8. Det som er viktig er å kunne lese et parition entry. En partition-entry er beskrevet i 11.9.

Adresse	beskrivelse	Lengde(bytes)
0x0	bootstrap code	446
0x1be	partition entry 1	16
0x1ce	partition entry 2	16
0x1de	partition entry 3	16
0x1ee	partition entry 4	16
0x1fe	boot signature	32

Figur 11.8: Master Boot Record

Adresse	beskrivelse	Lengde(bytes)
0x0	status	1
0x1	første adresse (CHS)	3
0x4	partisjontype	1
0x5	siste adresse (CHS)	3
0x8	første adresse (LBA)	4
0xc	antall sektorer	4

Figur 11.9: partition entry

11.2.2 Bios Parameter Block

Den partisjonen som skal lese er partisjon nummer 1. Deretter skal første adresse(LBA) leses. Logical Block Addressing (LBA) er et alternativ til den mer tradisjonelle Cylinder-Head-Sector (CHS) adresseringen som er en tidlig metode for å adressere hard-disker. LBA er mer logisk å bruke når lagringsenheten lagrer data som en lineær sekvens, slik som ved et SD-kort.

FAT Boot Record vil ligge i starten av den første partisjonen. Denne okkuperer en sektor og er alltid plassert ved den logiske adresse 0 av partisjonen. Boot Record ser ut som i figur 11.10. For å finne ut om det er FAT32, eller FAT12/16 kan man lese antall sektorer på enheten. Om dette tallet er 0 er det mer enn 65535 sektorer, og man må bruke FAT32 for å dekke alle sammen.

For å lese resten av PBP må man ta hensyn til noen forskjeller mellom FAT32 og FAT16/12. Dette er fordi det trengs flere byte på å beskrive et system med ord-størrelse 32 enn 16. Fortsettelsen av PBP FAT32 kan sees i figur 11.11.

Adresse	beskrivelse	Lengde(bytes)
0	magisk tall	3
3	OEM ID	8
11	antall bytes i en sektor	2
13	antall sektorer i en klyng	1
14	antall reserverte sektorer	2
16	antall FAT'er i partisjonen	1
17	Antall mapper	2
19	Antall sektorer i enheten	2
21	Media descriptor	1
22	Antall sektorer per FAT	2
24	Reservert	4
28	Antall skjulte sektorer	4
32	Stort antall sektorer på enheten	4

Figur 11.10: FAT BPB

Adresse	beskrivelse	Lengde(bytes)
36	FAT størrelse i sektorer	4
40	Flagg	2
42	Fat versjon	2
44	klyngernummer for Root mappen	4
48	sektornummer nummer for FSinfo strukturen	2
50	sektornummer for backup Boot Record	1
52	Reservert	12
64	Enhetsnummer	1
65	flagg for windows NT	1
66	signatur	1
67	Volum ID	4
71	Volum navn	4
82	System-id string(alltid "FAT32"	8
90	Boot kode	420
510	Bootable partisjon signatur	2

Figur 11.11: FAT32 Extended Boot Record

11.2.3 Initialisere FAT-modulen

For å initialisere modulen må den lese MBR, finne første partisjon, så lese BPB og finne ut hvilken FAT versjon det er. Etter dette kan den lese noen viktige verdier og regne ut hvor root-mappen ligger. På linje 3 leses første blokken i SD-kortet som inneholder MBR. Deretter finnes adressen til partisjon nummer 1 på linje 16 til 19. Grunnen til at verdien leses på denne måten er at verdien er lagret som little-endian. Deretter kan den lese første blokk av den første partisjonen som er BPB-en. Dette gjøres på linje 22.

Deretter finner funksjonen ut hvilken versjon av FAT som leses. Dette gjøres ved å regne ut hvor mange klynger det er på enheten. Om det er mer enn 65525 er det FAT32, mer enn 4085 er det FAT16 og mindre enn det er FAT12. Deretter lagres viktige verdier i en en variabel i minnet. Viktige variabler er hvor root-mappen ligger, hvor data-området begynner, hvor FAT området begynner, størrelsen på en klynge og sektor og størrelsen på en FAT.

```
1  int fat_init(struct fs **filesystem){
2      // read master boot record to find where the first partition begins
3      uint8_t buf[512];
4      int r = sd_read(buf, FAT_BLOCK_SIZE, 0);
5
6      if(r < 0){
7          uart_puts(''FAT: error reading mbr\r\n'');
8          return r;
9      }
10     if(r != 512){
11         uart_puts(''FAT: error reading mbr (only '');
12         uart_put_uint32_t(r, 10);
13         uart_puts('' bytes read)\r\n'');
14         return -1;
15     }
16     uint32_t partition_offset = buf[454] << 0
17                               |   buf[455] << 8
18                               |   buf[456] << 16
19                               |   buf[457] << 24;
20
21     //read block 0 where the boot section is
22     r = sd_read(buf, FAT_BLOCK_SIZE, partition_offset);
23     if(r < 0){
24         uart_puts(''FAT: error reading fat block 0\r\n'');
25         return r;
26     }
27     if(r != 512){
28         uart_puts(''FAT: error reading fat block 0 (only '');
29         uart_put_uint32_t(r, 10);
30         uart_puts('' bytes read)\r\n'');
31         return -1;
32     }
33     struct fat_BS myfat;
34     copy_to_fat_BS(&myfat, buf);
35
36     if(myfat.bootjmp[0] != 0xeb){
37         return -1;
```

```
38      }
39
40      uint32_t total_sectors = myfat.total_sectors_16;
41      if(total_sectors == 0){
42          total_sectors = myfat.total_sectors_32;
43      }
44      struct fat_fs *ret = (struct fat_fs *)malloc(sizeof(struct fat_fs));
45
46      memset(ret, 0, sizeof(struct fat_fs));
47
48      ret->b.fs_load = fat_load;
49      ret->b.fs_store = fat_store;
50      ret->total_sectors = total_sectors;
51
52      ret->bytes_per_sector = (uint32_t)myfat.bytes_per_sector;
53      ret->root_dir_entries = myfat.root_entry_count;
54      ret->root_dir_sectors = (ret->root_dir_entries * 32
55              +  ret->bytes_per_sector -1) / ret->bytes_per_sector;
56
57      uint32_t size = myfat.table_size_16;
58      if(size == 0){      // is fat32
59          copy_to_fat32(&(myfat.ext.fat32), buf);
60          size = myfat.ext.fat32.table_size_32;
61      }
62
63      uint32_t data_sec = total_sectors - (myfat.reserved_sector_count + myfat.table_count * size
64                                           + ret->root_dir_sectors);
65
66      uint32_t total_clusters = data_sec / myfat.sectors_per_cluster;
67
68      if(total_clusters < 4085){
69          ret->fat_type = FAT12;
70      }else if(total_clusters < 65525){
71          ret->fat_type = FAT16;
72      }else{
73          ret->fat_type = FAT32;
74      }
75      ret->b.fs_name = fat_names[ret->fat_type];
76      ret->sectors_per_cluster = myfat.sectors_per_cluster;
77      ret->bytes_per_sector = myfat.bytes_per_sector;
78      ret->vol_label = (char*)malloc(12);
79      // if fat32
80      if( ret->fat_type == FAT32){
81          copy_to_fat32(&(myfat.ext.fat32), buf);
82          strcpy(ret->vol_label, myfat.ext.fat32.volume_label);
83          ret->vol_label[11] = 0;
84          ret->first_data_sector = partition_offset + myfat.reserved_sector_count
85                          + (myfat.table_count * myfat.ext.fat32.table_size_32);
86          ret->first_fat_sector = myfat.reserved_sector_count + partition_offset;
87          ret->first_non_root_sector = ret->first_data_sector;
88          ret->sectors_per_fat = myfat.ext.fat32.table_size_32;
89          ret->root_dir_cluster = myfat.ext.fat32.root_cluster;
90      }else{ // if fat16/12
```

```
91          copy_to_fat16(&(myfat.ext.fat16), buf);
92          strcpy(ret->vol_label, myfat.ext.fat16.volume_label);
93          ret->vol_label[11] = 0;
94          ret->first_data_sector = partition_offset + myfat.reserved_sector_count +
95                  (myfat.table_count * myfat.table_size_16);
96          ret->first_fat_sector = myfat.reserved_sector_count + partition_offset;
97          ret->sectors_per_fat = myfat.table_size_16;
98
99          ret->root_dir_entries = myfat.root_entry_count;
100         // The + bytes_per_sector - 1 rounds up the sector no
101         ret->root_dir_sectors = (ret->root_dir_entries * 32
102                         + ret->bytes_per_sector - 1) /
103                 ret->bytes_per_sector;
104         ret->first_non_root_sector = ret->first_data_sector;
105     }
106     *filesystem = (struct fs*)ret;
107     return 1;
108 }
```

11.2.4 Lese en mappe

Nå vet modulen en del informasjon om FAT systemet. Når den skal lese en fil må den først lese root-mappen, og så gå videre fra der. En mappe ligger i data-området og er representert som en liste, der hver oppføring i lista er i formatet beskrevet i 11.12. Spesielle verdier i navnet gir informasjon om oppføringen dersom den ikke er i bruk. Om det første verdien i navnet er 0 er oppføringen ledig, og alle oppføringer etter denne er også ledig. Om verdien er 0x2e er det en "." eller ".." mappe. Om det er 0xe5 var det tidligere en oppføring her, men den ble slettet. Mapper kan, i likhet med filer, være større enn en klynge, og dermed være spredd over flere klynger.

Dette vil si at det er ganske enkelt å lese en mappe. Man trenger bare å iterere over disse verdiene og returnere dem. Dette gjøres i funksjonen fat_read_dir.

Offset	beskrivelse	Lengde(byte)
0x0	Filnavn	11
11	Attributter	1
12	Reservert	1
13	Tidspunktet filen ble laget	5
18	Tidspunktet filen sist ble aksessert	2
20	De 16 mest signifikante bitene i oppføringens første klynge	2
22	Siste gang filen ble modifisert	4
26	De minst signifikante bitene i oppføringens første klynge	2
28	Filstørrelsen i byte	4

Figur 11.12: En oppføring i en mappe

Fat_read_dir. tar inn en dirent variable som inneholder hvor man finner en mappe eller fil. Om dens verdi er NULL leser funksjonen root-mappen i stedet. på linje 19 til 31 leser den inn klyngen som *dir gav. Deretter itererer den gjennom området og tolker mappe-oppføringene. Om det er en fil eller mappeoppføring lager funksjonen en dirent node, og legger den i en lenket liste. Når funksjonen har iterert seg gjennom hele klyngen prøver den å få tak i neste klynge. Dette gjøres i funksjonen

get_next_fat_entry. Dette er en enkel funksjon som sjekker om den nåværende klyngens FAT indeks har en peker videre til neste klynge. Om neste klynge har en verdi større enn 0x0ffffff7 er det ikke en neste klynge. Om det er en neste klynge starter denne løkken på nytt igjen.

```
1  struct dirent *fat_read_dir(struct fat_fs *fs, struct dirent *dir){
2      int root = 0;
3      struct fat_fs *fat = fs;
4      if(dir == NULL){
5          root = 1;
6      }
7      uint32_t cur_cluster;
8      uint32_t cur_root_cluster_offset = 0;
9      if(root){
10         cur_cluster = fat->root_dir_cluster;
11     }else{
12         cur_cluster = dir->cluster_no;
13     }
14     struct dirent *ret = NULL;
15     struct dirent *prev = NULL;
16
17     do{
18         // Read this cluster
19         uint32_t cluster_size = fat->bytes_per_sector * fat->sectors_per_cluster;
20         uint8_t *buf = (uint8_t *)malloc(cluster_size);
21
22         // Interpret the cluster number to an absolute address
23         uint32_t abs_cluster = cur_cluster - 2;
24         uint32_t first_data_sector = fat->first_data_sector;
25         if(!root){
26             first_data_sector = fat->first_non_root_sector;
27         }
28         int br_ret = sd_read( buf, cluster_size,
29                 abs_cluster * fat->sectors_per_cluster + first_data_sector);
30         if(br_ret < 0){
31             return (void*)0;
32         }
33
34         for(uint32_t ptr = 0; ptr < cluster_size; ptr += 32){
35             // exists?
36             if((buf[ptr] == 0) || (buf[ptr] == 0xe5)){  continue;}
37             // the ''.'' or ''..'' directory?
38             if(buf[ptr] == '.')   {continue;}
39             // long filename entry?
40             if(buf[ptr + 11] == 0x0f){  continue;}
41
42             // if not, we can read the entry
43             struct dirent *de = (struct dirent *)malloc(sizeof(struct dirent));
44             memset(de, 0, sizeof(struct dirent));
45             if(ret == (void *)0)
46                 ret = de;
47             if(prev != (void *)0)
48                 prev->next = de;
49             prev = de;
```

```
50
51              de->name = (char *)malloc(13);
52              // interpret name
53              int name_index = 0;
54              int ext = 0;
55              int has_ext = 0;
56              for(int i = 0; i < 11; i++){
57                  char cur_v = (char)buf[ptr + i];
58                  if(i == 8){
59                      ext = 1;
60                      de->name[name_index++] = '.';
61                  }
62                  if(cur_v == ' ')
63                      continue;
64                  if(ext)
65                      has_ext = 1;
66                  if((cur_v >= 'A') && (cur_v <= 'Z'))
67                      cur_v = 'a' + cur_v - 'A';
68                  de->name[name_index++] = cur_v;
69              }
70              if(!has_ext)
71                  de->name[name_index - 1] = 0;
72              else{
73                  de->name[name_index] = 0;
74              }
75              if(buf[ptr + 11] & 0x10){
76                  de->is_dir = 1;
77              }
78              de->next = NULL;
79              de->byte_size = *(uint32_t*)&buf[ptr + 28];
80              uint32_t cluster_num = *(uint16_t*)&buf[ptr + 26]
81                      | ((uint32_t)(*(uint16_t*)&buf[ptr + 20]) << 16);
82              de->cluster_no = cluster_num;
83          }
84          free(buf);
85          // next cluster
86          if(root && (fs->fat_type != FAT32)){
87              cur_root_cluster_offset++;
88              if(cur_root_cluster_offset < (fat->root_dir_sectors /
89                      fat->sectors_per_cluster)){
90                  cur_cluster++;
91              }else{
92                  cur_cluster = 0x0ffffff8;
93              }
94          }
95          else{
96              cur_cluster = get_next_fat_entry(fat, cur_cluster);
97          }
98      }while(cur_cluster < 0x0ffffff7);
99      return ret;
100 }
```

11.2.5 Lese en fil

Funksjonen som blir kalt for å lese en fil er fat_read. Den tar klyngenummeret til en fil og en buffer. Den er en ganske enkel funksjon som bare leser fra klyngen gitt som argument, og skriver det i bufferen. Før loopen begynner regner den ut størrelsen på en klynge. Deretter går den inn i while-løkken som først finner ut hvilken sektor som skal lastes, så finner den ut hvor stor lese-buffer som trengs. Deretter kalles sd_read som er forklart i 11.1.3. Dette gjøres på linje 10 til 13. Deretter må funksjonen overføre den leste data fra lese-bufferen til bufferen gitt som argument (buf). I if setningen på linje 20 til 37 finner funksjonen ut hvor i argument-bufferen funksjonen skal koiere til, hvor i lese-bufferen den skal kopiere fra og hvor langt den skal kopiere. Etter å ha gjort en iterasjon leser bruker den get_next_fat_entry for å finne neste klynge til filen.

```
1  static size_t fat_read(struct fat_fs *filesystem, uint32_t cluster_no, uint8_t *buf,
2                         size_t byte_count, size_t offset){
3      uint32_t cur_cluster = cluster_no;
4      size_t cluster_size = filesystem->bytes_per_sector * filesystem->sectors_per_cluster;
5      size_t file_loc = 0;
6      int buf_ptr = 0;
7      while(cur_cluster < 0x0ffffff8){
8          if(( file_loc + cluster_size) > offset){
9              if( file_loc < (offset + byte_count)){
10                 //  load this cluster, as it contains the requested file
11                 uint32_t sector = get_sector(filesystem, cur_cluster);
12                 uint8_t * r_buf = (uint8_t*)malloc(cluster_size);
13                 int ret = sd_read(r_buf, cluster_size, sector);
14                 // check for error
15                 if(ret < 0){
16                     return ret;
17                 }
18                 int len;
19                 int c_ptr;
20                 if(offset >= file_loc){
21                     // first cluster of file
22                     buf_ptr = 0;
23                     c_ptr = offset - file_loc;
24                     len = byte_count;
25                     if(len > (int)cluster_size){
26                         len = cluster_size;
27                     }
28                 }else{
29                     // not first cluster of file
30                     c_ptr = 0;
31                     len = byte_count - buf_ptr;
32                     if(len > (int)cluster_size)             { len = cluster_size;}
33                     if(len > (int)(byte_count - buf_ptr))   { len = byte_count - buf_ptr;}
34                 }
35                 memcpy(buf + buf_ptr, r_buf + c_ptr, len);
36                 free(r_buf);
37                 buf_ptr += len;
38             }
39         }
40         cur_cluster = get_next_fat_entry( filesystem, cur_cluster);
```

```
41          file_loc += cluster_size;
42      }
43      return buf_ptr;
44  }
```

11.2.6 Lese fra fa filsystemet

Funksjonen som blir brukt av brukeren som skal bruke modulen er fat_load. Denne funksjonen tar inn en path til filen og en buffer. Det første funksjonen gjør er å finne ut hvor filen ligger. Dette gjør den i while-løkken på linje 12 til 43. Denne funksjonen finner først det første segmentet av path. Om path er, for eksempel, "folder1/folder2/myfile.elf", vil det første path-segmentet være folder1. Deretter kaller den fat_read_dir med nåværende mappe som argument. Fat_read_dir er beskrevet i 11.2.4. I den første iterasjonen vil den nåværende node være NULL, dermed lese root-mappen. Funksjonen returnerer en lenket lite med dirent variabler som tilsvarer filer eller mapper.

Deretter itererer funksjonen seg gjennom disse nodene på linje 23 til 34. Om navnet på mappen/filen er likt path-segmentet tas den ut, resten slettes. Om path-segmentet ikke ble fant avsluttes funksjonen, og gir en feilverdi tilbake. Til slutt i løkken sjekker den om dette er filen som funksjonen leter etter. Om dette er filen avslutter løkken. Deretter kalles fat_read for å laste filen inn i bufferen gitt som argument.

```
1   int fat_load(const char *path, uint8_t *buf, uint32_t buf_size){
2       const char *c = path;
3       int offset;
4       struct dirent *node = NULL;
5       struct dirent *current_node = NULL;
6       struct dirent *free_node = NULL;
7       struct fs* filesys = fs_get();
8       if( path == NULL){  return -1; }
9
10      // find first folder/file
11      // look for '/' or '/'
12      while(1){
13          if(*c == '/'){ c++;}
14          offset = fat_get_next_path(c);
15          char* path_buf = (char*)malloc(sizeof(char) * offset + 1);
16          memcpy(path_buf, c, offset);
17          path_buf[offset] = '\0';
18          // now path_buf contains the next segment in the path.
19          // search for it!
20          // delete all the other nodes in the linked list
21          node = fat_read_dir( (struct fat_fs*)filesys, current_node);
22          while(node){
23              if( !strcmp( node->name, path_buf) ){
24                  current_node = node;
25                  node = node->next;
26              }else{
27                  free_node = node;
28                  node = node->next;
29                  free(free_node);
30              }
```

```
31          }
32          if( current_node == NULL)    { return -1; }
33          // current_node now contains the dirent for the next path segment
34          if(current_node->is_dir){
35          }else{
36              break;  // is file
37          }
38          c = c + offset;
39          free(path_buf);
40      }
41      //  do something to load the file
42
43      if( current_node->byte_size > buf_size){
44          uart_put_uint32_t(current_node->byte_size, 10);
45          return -1;
46      }
47
48      int ret = fat_read(     (struct fat_fs*)filesys,
49                              current_node->cluster_no,
50                              buf,
51                              current_node->byte_size,
52                              0);
53
54      return ret;
55  }
```

12 Postboksen

For at KybOS skal bruke alle fire prosessorene og kommunisere med GPU-en er det nødvendig å implementere en postboks funksjonalitet i kjernen. En postboks fungerer ved at hver prosessor og GPU har input-output registre som definerer et grensesnitt mellom dem. Dette grensesnittet kan lese om på raspberry pi sin github wiki, [6]. Disse registrene er organisert slik:

```
typedef struct{
    volatile uint32_t   READ;
    volatile uint32_t   unused[3];
    volatile uint32_t   PEEK;
    volatile uint32_t   SENDER;
    volatile uint32_t   STATUS;
    volatile uint32_t   CONFIG;
    volatile uint32_t   WRITE;
} mailbox_t;
```

Grensesnittet fungerer slik at sender skriver en adresse i WRITE registeret til motagende postboks. Da vil motagende enhet få et avbrudd og kan lese fra denne adressen. Denne adressen inneholder en buffer. Bufferen må være 0x100 justert (dokumentasjonen sier 0x10 justert, men dette må være en skrivefeil da dette gir feil), og inneholder verdier som gir mening til mottager. Hva som skjer videre er helt opp til implementasjonen.

12.1 Kommunisere med GPU

Det er allerede implementert et grensesnitt for GPU-en. Det er definert et sett med kommandoer som vil få GPU-en til å utføre en handling, og så skrive en retur-melding ved å overskrive den sendte meldingen. Den sendende enhet kan nå lese retur meldingen.

Dette er implementert slik i KybOS:

```
#define MAILBOX_STATUS_EMPTY    0x40000000
#define MAILBOX_STATUS_FULL     0x80000000
#define MAILBOX_VM_OFFSET       0x40000000
void mailbox_write(volatile uint32_t *addr, uint8_t chan){
    while( mailbox_vc_get()->STATUS & MAILBOX_STATUS_FULL){} // do nothing
    mailbox_vc_get()->WRITE = (chan | (uint32_t)addr | MAILBOX_VM_OFFSET);
}

```

```
9  uint32_t mailbox_read(uint8_t chan){
10     uint32_t data;
11     while(1){
12         while( mailbox_vc_get()->STATUS & MAILBOX_STATUS_EMPTY){ /* do nothing*/ }
13         data = mailbox_vc_get()->READ;
14         if (chan == (uint8_t)(data & 0xf)){
15             return (data & 0xfffffff0);
16         }
17     }
18 }
19
20 // for flushing the buffer.
21 static void flush(void){
22     while(!( mailbox_vc_get()->STATUS & MAILBOX_STATUS_EMPTY)){
23         mailbox_vc_get()->READ;
24     }
25 }
26
27 uint32_t mailbox_write_read(volatile uint32_t *addr, uint8_t chan){
28     mmu_cache_invalidate((uint32_t)mailbuffer);
29     barrier_data_mem();
30     flush();        // make sure buffer is empty
31     mailbox_write(addr, chan);
32     barrier_data_mem();
33     uint32_t result = mailbox_read(chan);
34     barrier_data_mem();
35     return result;
36 }
```

Write vil skrive en adresse som inneholder sendende melding. Den sjekker først om postboksen til GPU-en allerede er full. Da må den vente til den er tom. Deretter skriver den til WRITE registeret. Her er det viktig å justere for at CPU-en og GPU-en på Raspberry Pi 2 er minne mappet forskjellig, slik at det må justeres for. Denne justeringsverdien er 0x40000000 når hurtigbufferen er på, i følge [10].

Read er en enkel vente-funksjon som brukes etter å ha skrevet en verdi. Den bruker en while-løkke til å vente til GPU-en har skrevet over meldingen med en retur-melding. Retur verdien, data, skal være adressen til bufferen.

funksjonen write_read samler disse to funksjonene. Den bruker data barriere for å sikre at alle dataaksessene er ferdig når neste fase begynner. Det første funksjonen gjør er å invalidere hurtigbuffer-området til bufferen. Dette er fordi at med innstillingene til hurtigbufferen write_back (se kapittel 6.1.1) vil ikke bufferen skriver til minne, bare til hurtigbufferen. Dette er selvfølgelig katastrofalt fordi GPU-en bare lese søppel. Dette fikses med å invalidere hurtigbuffer området til bufferen slik at det skrives til minnet. Deretter kalles write, deretter read før det returnerer med adressen til bufferen. Denne returverdien er strengt tatt ikke nødvendig i dette tilfellet fordi bufferen er skrivet over, så adressen er allerede kjent.

13 Arkitektur

Dette kapittelet vil omhandle KybOS' arkitektur. Målet er å planlegge en måte å kunne distribuere KybOS over flere noder slik at det for prosesser ser ut som ett system. Det er viktig at det finnes grensesnitt som operativsystemet kan benytte for å gjøre viktige operasjoner. Disse protokollene må bli implementert i en nettverksmodul som gjør det enkelt for systemet å bruke grensesnittet. Det må også implementeres en nettverksdriver som driver enheten som det kommuniserer over. Et godt alternativ er å for eksempel bruke en ethernet forbindelse for å kommunisere over Internett. Viktige sekvenser i grensesnittet er:

Tilkobling En node kobler seg til et nettverk av flere noder.

Finne prosess En node prøver å finne en navngitt prosess.

Finne driver En node vil ha en driver som ikke finnes på sin node, og må spørre andre noder.

13.1 Tilkobling

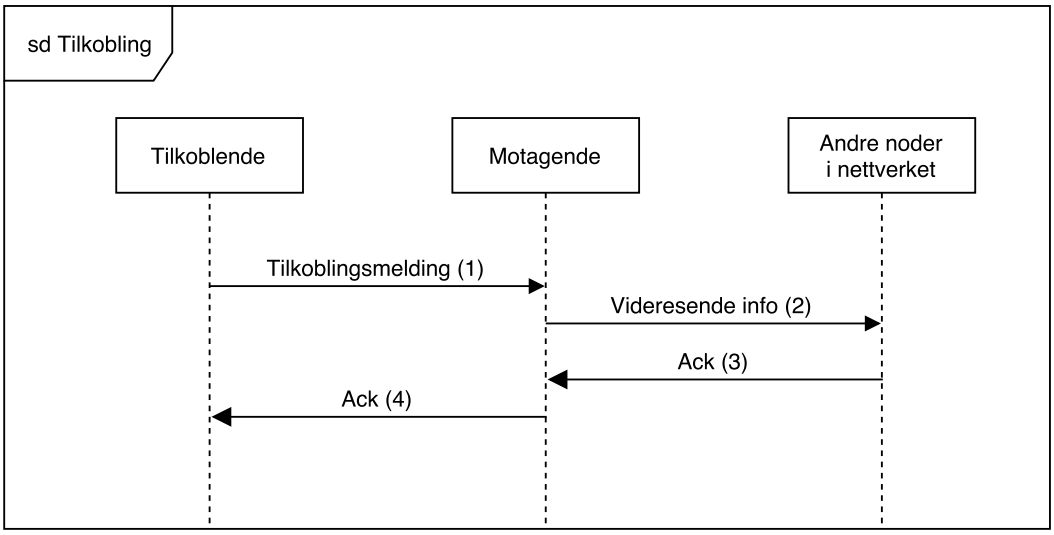
Når en node har gjort en oppstart kan noden helt fint kjøre for seg selv. Da vil noden kjøre som et mikrokjerne operativsysem på en enkelt node. Om man ønsker å bruke den som en del av et distribuert system må nodene vite om hverandre og hvordan man kan kontakte dem. Node som skal koble seg til nettverket må sende en tilkoblingsmelding til en IP-adresse som den vet er en node med KybOS på. Denne meldingen inneholder informasjon om noden som trengs for å jobbe i nettverket. Sekvensen skjer slik:

1. Tilkoblende node sender en tilkoblinsmelding til en node som den vet inneholder KybOS.

2. Den mottagende sender informasjonen i denne meldingen videre til alle i nettverket. Informasjon som må sprese er for eksempel IP-adressen.

3. Alle de andre nodene i nettverket sender en ack-melding tilbake.

4. Når alle ack-meldingene har kommet sendes en ack-melding tilbake til tilkoblende node som et signal om at den er nå i et nettverk. Denne ack-meldingen inneholder også informasjon om alle de andre nodene i nettverket som trengs for å kommunisere med dem. Dette inkluderer IP-adressene til alle nodene som er i nettverket.

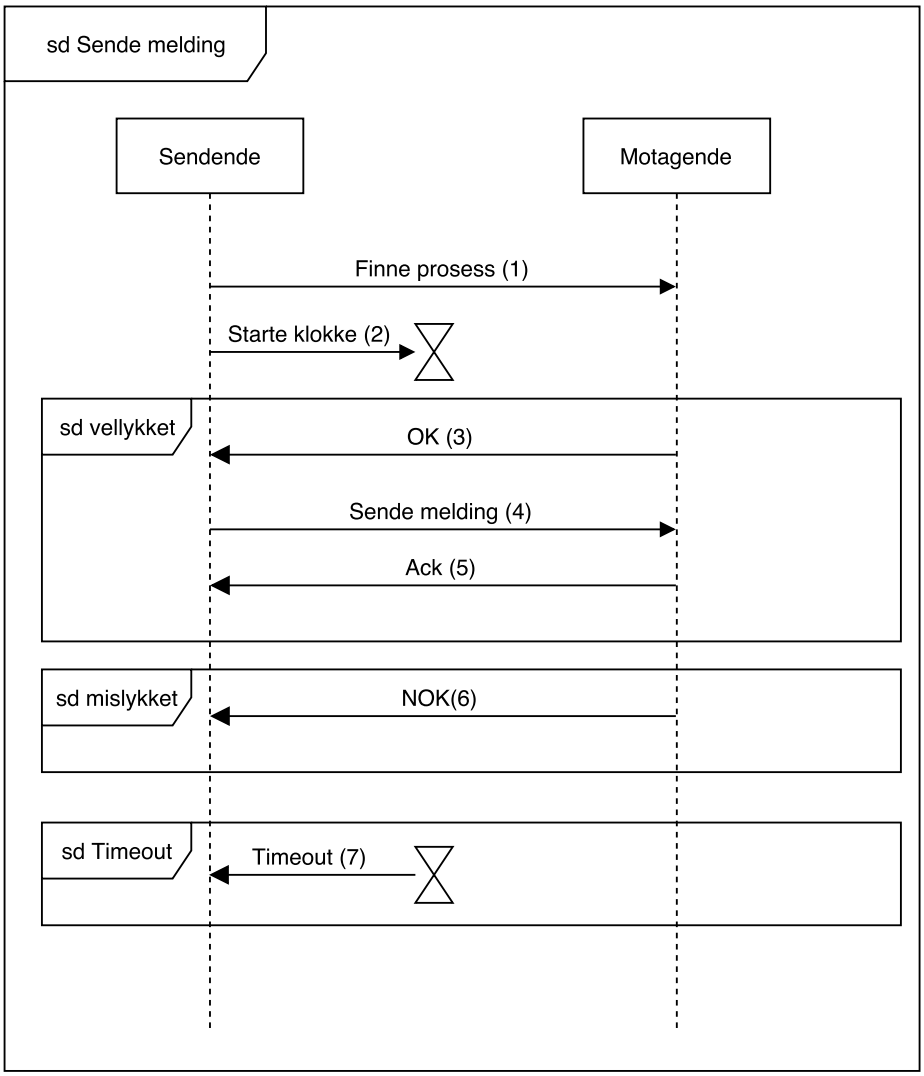


Figur 13.1: Sekvensdiagram av tilkobling

13.2 Finne en prosess

Når prosesser skal sende meldinger til hverandre er det ikke sikker at den mottagende prosess er på samme node som sendende prosess. Da må noden prøve å få tak i denne prosessen over nettverket. Som beskrevet i 8.1 har identiteten til en prosess IP-adressen til noden i seg. Dette gjør det svært enkelt på finne prosessen. Man kan sende en melding til IP-adressen i identiteten til prosessen og spørre om denne prosessen er der. Dermed kan sekvensen for å sende en melding til en annen node se ut som i figur 13.2.

1. Sendende node sender en melding til motagende node for å høre om prosessen eksisterer på denne noden.
2. Sendende node starter også en klokke som kan gi timout om det tar for lang tid.
3. Om prosessen fins på denne noden sender noden tilbake en OK melding.
4. Da kan sendende node sende sendende prosess' melding til mottagende node, som vil gi den videre til riktig prosess.
5. Mottagende node sender en ack-melding tilbake for å si at alt gikk bra.
6. Om mottagende node ikke finner prosessen den blir spurt om vil den sende en NOK-melding tilbake.
7. Om mottagende node ikke svarer eller bruker for lang tid vil klokken gi en timeout, som stopper sekvensen.

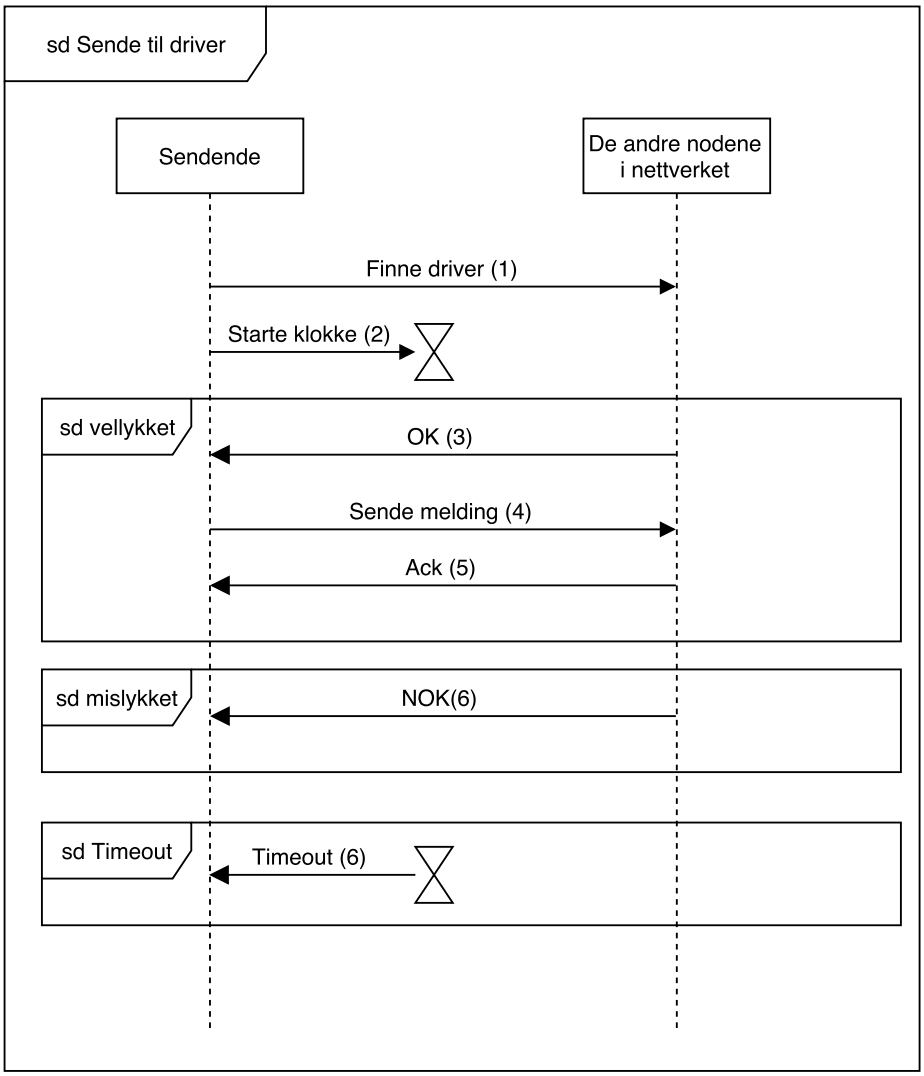


Figur 13.2: Sekvensdiagram av å sende melding til en annen node

13.3 Finne en driver

Når en prosess vil finne den driver er det ikke linke enkelt som å finne en prosess. En driver blir gitt ved et navn, for ekstempel ”gpio”. Om ”gpio” driveren ikke finnes på noden må noden spørre alle andre noded i nettverket om de har en ”gpio” driver. Denne sekvensen ligner veldig på den i 13.2. Forskjellen er at melingen må sendes til alle nodene i nettverket, ikke bare til én.

1. Sendende node sender en melding til alle node for å høre om driveren eksisterer på denne noden.
2. Sendende node starter også en klokke som kan gi en timeout om det tar for lang tid.
3. Om driveren fins på denne noden sender noden tilbake en OK melding.
4. Da kan sendende node sende sendende prosess’ melding til mottagende node, som vil gi den videre til riktig driver.
5. Mottagende node sender en ack-melding tilbake for å si at alt gikk bra.
6. Om mottagende node ikke finner driveren den blir spurt om vil den sende en NOK-melding tilbake. Om alle noder sender tilbake NOK-mmeldinger kan sendende noe konkludere at driveren ikke eksistere på noen noder.
7. Om mottagende node ikke svarer eller bruker for lang tid vil klokken gi en timeout, som stopper sekvensen.



Figur 13.3: Sekvensdiagram av å sende melding til en driver på en annen node

14 Tester

I dette kapittelet testes sanntidsegenskapene til KybOS. Det er gjennomført 2 tester. En test som måler tiden det tar å gjøre en kontekst bytte fra en prosess til en annen. Den andre testen er en test som måler hva responstiden til systemet er.

14.1 kontekstbytte test

Formålet med denne testen var å måle hvor lang tid et kontekst bytte tar. Dette ble utført ved å måle antall kontekstbytter i løpet av ett sekund. Dette ble gjort ved å sette inn denne koden i starten av context_switch_c.

```
1  uint32_t context_switch_c(uint32_t old_sp){
2      static time_unit_t time2;
3      time_unit_t time1;
4      static int count = 0;
5      if( test == 1){
6          if( count == 0){
7              uart_puts("starting test\r\n");
8              time2 = time_get();
9              time_add_microseconds(&time2, 1000);
10         }
11         count++;
12         time1 = time_get();
13         if( time_compare(time1, time2) == 1){
14             uart_puts("test done\r\n");
15             uart_puts("count: ");
16             uart_put_uint32_t(count, 10);
17             while(1){
18                 /* nothing */
19             }
20         }
21
22     }
23     ...
24 }
```

Når systemet har initialisert seg, ble gpio-avbrudd brukt til å gjøre to ting. Det ene er å sette variabelen test lik en, for å starte testen. Den andre er å produsere avbrudd hele tiden så det eneste som kjernen gjør er å gjøre kontekst bytter. I initialiseringen, på linje 6 til 10 finner funksjonen den nåværende

tiden, og legger på ett sekund. Dette legges i variabelen time2. Når kjernen videre utfører kontekst bytter øker den count med en, finner nåværende tid og legger det i time1, og sammenligner time1 og time2 i en if setning. Om time1 er større enn time2 har det gått over ett sekund siden testen starte. Da printes variabelen count på terminalen.

Resultatene var stabile. Over 5 tester var resultatet at det ble gjort 60520, 60555, 60504, 60502 og 60534 kontekstbytter i løpet av ett sekund. Gjennomsnittet av dette er 60523. Dette betyr at ett kontekstbytte tar omtrent $1s/60534Hz \approx 16.5\mu s$.

14.2 Responstest

Her ble det koblet opp en responstester til raspberry pi-en. Denne responstesteren har fire output linjer og fire input linjer. Den fungerer slik at den setter et output-linjen høyt, og venter til den tilsvarende input-linjen blir satt høyt. Ved å måle tidsforskjellen klarer den å måle responstiden til noe som er koblet til. Dette er beskrevet i [11]. De fire output signalene ble koblet til GPIO-pins på raspberry PI-en som var konfigurert til input og til å gi avbrudd på flanke opp, og har pull-down motstander. De fire input linjene ble koblet til GPIO-pins som var konfigurert som input.

KybOS som kjørte var konfigurert slik at det var to prosesser. En GPIO-driver som konfigurerte GPIO-pins-ene til å fungrere som beskrevet over. Den kjørte med høy prioritet slik at den alltid vil kjøres når den kan. Denne er implementert slik:

```
1  void delay(int32_t count){
2      __asm volatile(''__delay_%=: subs %[count], %[count], #1;
3              bne __delay_%=\n'' :: [count]''r''(count) : ''cc'');
4  }
5
6  int main(void){
7      int flags = 0;
8      process_id_t sender;
9      driver_register("gpio");
10     volatile uint32_t *gpfsel4 = mmap(GPFSEL4);
11     volatile uint32_t *gpfsel2 = mmap(GPFSEL2);
12     volatile uint32_t *gpfsel1 = mmap(GPFSEL1);
13     volatile uint32_t *gpfsel0 = mmap(GPFSEL0);
14     volatile uint32_t *gpset0  = mmap(GPSET0);
15     volatile uint32_t *gppud   = mmap(GPPUD);
16     volatile uint32_t *gppudclk0 = mmap(GPPUDCLK0);
17     volatile uint32_t *gplev0  = mmap(GPLEV0);
18     volatile uint32_t *gpclr0  = mmap(GPCLR0);
19     volatile uint32_t *gpren0  = mmap(GPREN0);
20
21     /**********INPUT PINS*********/
22     *gpfsel2 &= ~(7 << 18); //  pin 26
23     *gpfsel1 &= ~(7 << 29); //       19
24     *gpfsel2 &= ~(7 << 9);  //       13
25     *gpfsel0 &= ~(7 << 18); //       6
26
27     // enable pull-down resistors on input pins
28     *gppud = 1;
29     delay(300);
```

```
30      *gppudclk0 = (1 << 26);
31      *gppudclk0 = (1 << 19);
32      *gppudclk0 = (1 << 13);
33      *gppudclk0 = (1 << 6);
34      delay(300);
35      *gppud = 0;
36      *gppudclk0 = 0;
37
38      // enable rising edge interrupts
39      *gpren0 |= (1 << 26);
40      *gpren0 |= (1 << 19);
41      *gpren0 |= (1 << 13);
42      *gpren0 |= (1 << 6);
43
44      /**********OUTPUT PINS*************/
45
46      // zero pin
47      *gpfsel2 &= ~(7 << 3);   //  pin 21
48      *gpfsel2 &= ~(7 << 0);   //      20
49      *gpfsel1 &= ~(7 << 18);  //      16
50      *gpfsel1 &= ~(7 << 6);   //      12
51      // write as output
52      *gpfsel2 |= (1 << 3);    //  pin 21
53      *gpfsel2 |= (1 << 0);    //      20
54      *gpfsel1 |= (1 << 18);   //      16
55      *gpfsel1 |= (1 << 6);    //      12
56
57      // set output level
58      *gpclr0 = ( 1 << 21);    // pin  21
59      *gpclr0 = ( 1 << 20);    // pin  20
60      *gpclr0 = ( 1 << 16);    // pin  16
61      *gpclr0 = ( 1 << 12);    // pin  12
62
63      // buf to receive interupts
64      while(1){
65          sender = ipc_receive(NULL, 0, &flags);
66          if(flags & BUF_TOO_SMALL){
67              _SYSTEM_CALL(4, (void*)"GPIO:␣buf␣too␣small\r\n", NULL, NULL);
68          }
69          if( *gplev0 & (1 << 26)){
70              *gpset0 = ( 1 << 21);    // pin  21
71              *gpclr0 = ( 1 << 21);    // pin  21
72          }
73          if( *gplev0 & (1 << 19)){
74              *gpset0 = ( 1 << 20);    // pin  21
75              *gpclr0 = ( 1 << 20);    // pin  21
76          }
77          if( *gplev0 & (1 << 13)){
78              *gpset0 = ( 1 << 16);    // pin  21
79              *gpclr0 = ( 1 << 16);    // pin  21
80          }
81          if( *gplev0 & (1 << 6)){
82              *gpset0 = ( 1 << 12);    // pin  21
```

```
83              *gpclr0 = ( 1 << 12);    // pin  21
84          }
85      }
86  }
```

Det første driveren gjør er å registrere seg selv som gpio-driveren. Dette fører til at hver gang det skjer et avbrudd som har kilde i en GPIO-pin så vil denne prosessen få en melding fra operativsystemet. Deretter kaller den mmap for å få tilgang på en del registre som ligger i minnet. Operativsystemet konfigurerer MMU-en, og returnerer en peker som vil peke på en virtuell adresse som blir oversatt til den fysiske adressen som mmap spør om. MMU-en blir også konfigurert slik at denne virtuelle adressen blir klassifisert som Device-minne. Dette er beskrevet i kapittel 6.1.1, 7.3 og 7.4.

Deretter begynner driveren å initialisere input-pins. Den setter pin 26, 19, 13, og 6 til å være input, ha pull-down motstander, og gi avbrudd på flanke-opp. Deretter initialiserer den fire pins, nummer 21, 20, 16 og 12 til å være output. Signalet til disse settes lav.

Til slutt går driveren inn i en evig while-løkke. Den mottar en melding ved å kalle ipc_receive. Dette kallet er blokkerende så prosessen stoppes intil den får en melding. Ipc_receive er beskrevet i 9.2.2. Når den mottar en melding vil driveren lese verdien på input-pins. Om verdien er satt høy på en av dem, vil driveren sette tilsvarende output pin høy i ett lite øyeblikk, og så lav. Dette vil responstesteren motta. Det som måles er altså tiden fra responstesteren setter et signal høyt, noe som genererer et GPIO-avbrudd, til GPIO-driveren setter tilsvarende output-pin høyt.

Den andre prosessen er en lavprioritets prosess som kjører i en evig while-løkke og gjør ingenting. Den kjører i bakgrunnen for at kjernen skal ha en prosess å kjøre når GPIO-driveren er blokkert på grunn av ipc_receive-kallet.

Sekvensen som forventes at skal bli gjennomført er som følger:

1. Responstester setter en linje høy og starter en klokke.
2. Bakgrunnsprosess kjører når kjernen mottar et avbrudd.
3. Kjernen finner ut at det var et GPIO-avbrudd, og sender derfor melding til GPIO-driveren.
4. Kjernen gjør et kontekst-bytte til GPIO-driveren.
5. Driveren setter riktig pin høy og så lav igjen.
6. Responstesteren mottar dette signalet og stopper klokken.

Med forbehold om at det er signalinjer, og at disse testes parallelt.

Ettersom det i kapittel 14.1 ble målt at et kontekst-bytte tar omtrent 16.5μs, kan en forvente at responstiden er noe høyere enn dette.

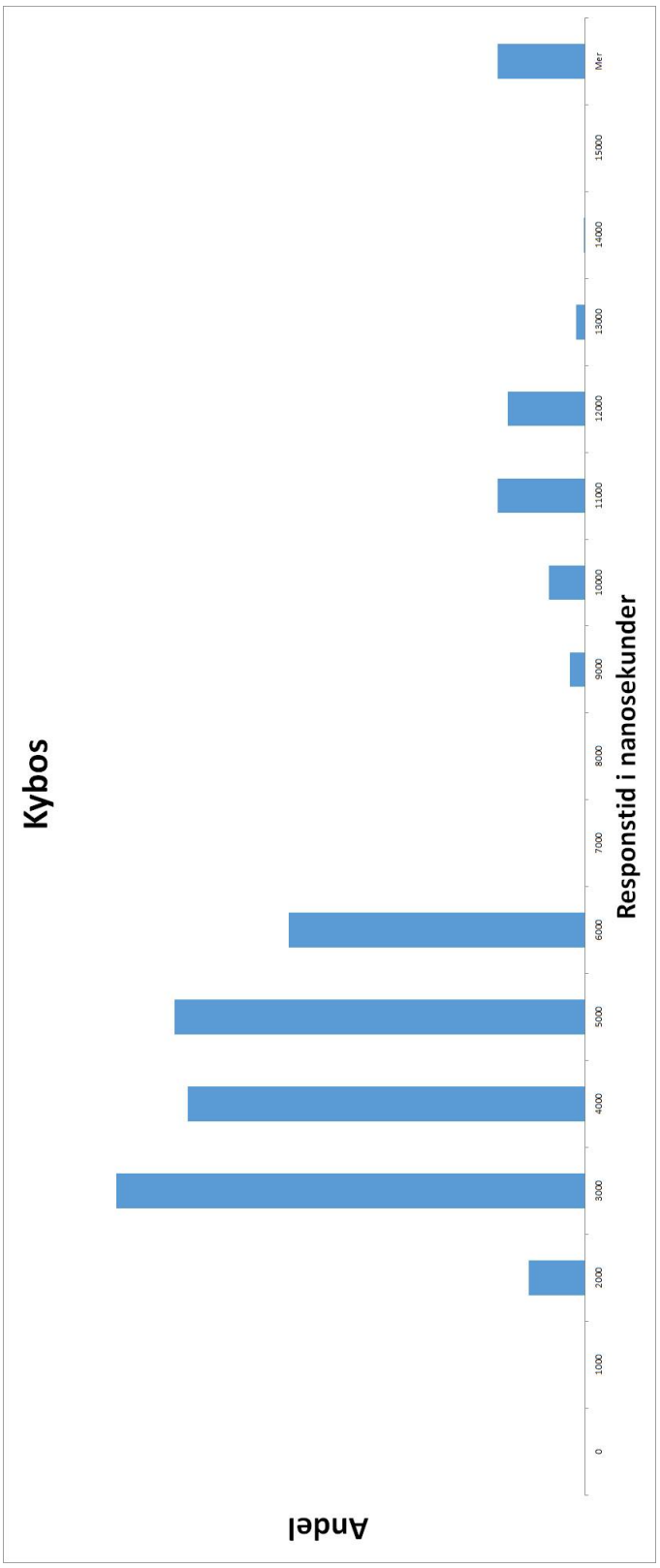
Resultatet er presentert i et histogram i figur 14.1. her kan man se at responstidene er sentrert rundt 3000 til 6000 nanosekunder. Det er også noen sentrert rundt 11000-1200 ns, samt en del i søylen ”mer”. Gjennomsnittstiden var omtrent 5400ns.

Dette var ikke nøyaktig som forventet. Ettersom det gjennomsnittlige kontekstbytte tar 16500 nanosekunder gikk dette mye raskere. Dette kan forklares med at driveren kan svare på flere signaler

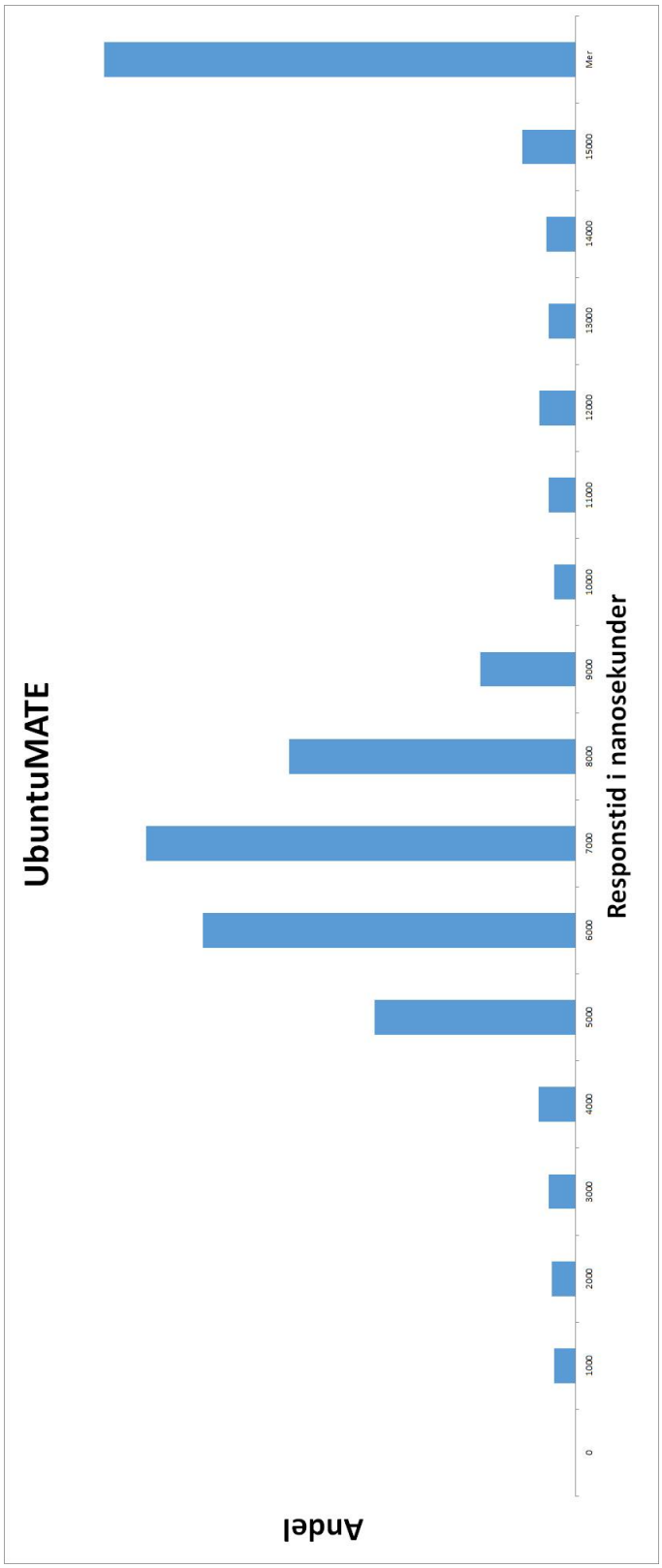
samtidig. høyden rundt 11000 til 12000 kan forklares ved at klokkeavbrudd skjer. Når ett klokkeavbrudd skjer vil systemet gjøre et kontekstbytte neste prosess i kø-en, også om det er seg selv som er den neste prosessen. Dette gjør at prosessen gjør et kontekst bytte til seg selv. Da vil systemet i effekt måtte gjøre 2 kontekst bytter før systemet får svart på signalet.

Kategorien Mer ser rar ut, og er overhodet ikke forventet. Det må komme av noe som skjer i kjernen og tar svært lang tid. Forfatteren er usikker på hva dette kan komme av, da det er tatt hensyn til sanntidsegenskaper under utviklingen. Amund Murstad sa også at dette kan stamme fra feil i responstesteren, så det er også en mulighet. Dette er uansett noe som burde bli sett på ved videre utvikling, da en slik tidsrespons er katastrofal for sanntidsegenskapene.

Systemet er likevel bra sammenlignet med andre operativsystem på Raspberry Pi 2. De mest populære er en versjon av linux. Amund Murstad har gjort lignende responstester på UbuntMate for Raspberry Pi 2. Denne testen ble gjort ved å kjøre 4 tråder på en prosessor som brukte polling til å lese signallinjene. Et histogram over resultatet kan ses i figur 14.2. Her kan en se at responstiden er sentrert rundt 7000 ns. Dette er noe tregere en KybOS. En kan også se at den har svært mye responstider spredt utover andre søyler. Den har også en svært stor søyle på ”Mer”. Gjennomsnittet var på 13400 ns. Dette er en mye dårligere responstid enn KybOS. Dette er som forventet da ubuntuMate ikke er laget med tanke på sanntidsegenskaper.



Figur 14.1: KybOS' responstid



Figur 14.2: Ubuntus responstid

15 Diskusjon

Målet ved dette prosjektet har vært å lage et mikrokjerne- og meldingsbasert operativsystem. Dette har i stor grad blitt oppnådd. En kjerne har blitt implementert som har gode egenskaper. Det var også et ønske om å kunne distribuere systemet. Det har ikke blitt implementert, men planlagt og tilrettelagt slik at fremtidige utviklere kan implementere denne funksjonaliteten

15.1 Hva har blitt gjort

I arbeidet er det gjort mange ting. Her er en liste over hovedpunktene.

1. KybOS klarer å bruke hardware på en fornuftig måte. Den gjennomfører oppstart av hardwareog C-runtime.
2. Muligheter for å feilsøke har blitt utforsket og implementert. Gode grensesnitt med UART og JTAG for å finne feil.
3. Har fått MMU-en til å fungere og laget funksjonalitet for å oversette virtuelt minne til fysisk minne. Ulike måter å konfigurere MMU-en har blitt undersøkt og det har blitt implementert en løsning som passer til KybOS sine krav.
4. Prosesser kan lages og kjøres. KybOS utfører fleroppgavekjøring slik at prosesser får en illusjon av å kjøre parallelt. Prosesser kjører i et eget adresserom. Dette involverer arbeid i fra MMU-en og flere avbrudshåndterere.
5. En postkasse funksjonalitet har blitt implementert for kommunikasjon med GPU-en og de andre prosessorene på kortet.
6. Det har blitt laget en SD-kort driver som gir funksjonalitet for å laste opp filer fra sd-kortet.
7. En enkel planlegger har blitt implementert.
8. Alle systemkallene som ble beskrevet i 2.5 har blitt implementert.
9. Det har blitt utført tester på hvor lang tid det tar å bytte kontekst, og responstid.

15.2 Resultater

KybOS har fått funksjonalitet for å kjøre prosesser. Disse prosessene kjører som om de kjøres parallelt ved hjelp av fleroppgavekjøring. Prosessene kan kommunisere med hverandre ved å sende melding

mer hverandre, synkronisert eller asynkront. Det er også implementert et drivergrensesnitt slik at prosesser kan betjene enheter. Systemkallene som ble beskrevet i 2.5 har blitt implementert på slik måte at de er robuste og fleksible.

Det har blitt gjennomført tester på sanntidsegenskaper. Resultatene av disse viser at KybOS er bedre en andre operativsystem på Raspberry Pi, men fortsatt ikke god nok til å ha gode sanntidsegenskaper.

Et svært viktig resultat av implementasjonen er at prosesser kjører i et virtuelt adresserom. Dette gjør at programmer kan kompileres separat fra kjernen. Etter at kjernen har startet kan disse programmet lastes inn i fysisk minne og kjøre i virtuelt minne. Den første 1MB av det virtuelle minnet er reservert for kjernen, ellers kan prosesser kjøre hvor som helst.

15.3 diskusjon

I dette prosjektet ble det gjort noen valg som burde vurderes i etterklokskapens navn.

- I kapittel 3 Platform så falt valget av platform på en Raspberry Pi 2. Dette viste seg å være et noe problematisk valg. For det først fantes det mangelfull dokumentasjon. Det fantes bare dokumentasjon om Raspberry Pi 1 sin mikrokontroller BCM2835, mens dokumentasjonen om BCM2836 var nærmest inteteksisterende. Derfor tok det lengre tid enn nødvendig å finne ut hvor minnelokasjonene til UART, klokker og avbruddskontrolleren var.

  Det var også noe mer komplisert å starte opp systemet og å få igang et skikkelig verktøyskjede på grunn av en feil i standard skriptet til arm-none-eabi-gcc. Dette var ikke hjulpet av at det også var et mindre online community for ”bare metal” applikasjoner enn tidligere trodd, og det som fantes var stort sett for Raspberry Pi 1. Dette gjorde at å finne eksempelkode var vanskeligere enn anslått.

  Her må det tas noe selvkritikk ved valget av platform. Likevel var ikke en Rasperry Pi 2 et dårlig valg, da når systemet først var oppe og kjørte fungerte det veldig bra, og passet alle andre kriterier, og var svært enkel å få tak i.

- I kapittel 5 ble det først implementert et grensesnitt mot PC. Senere viste det seg at behovet for å feilsøke med kraftigere hjelpemidler var stort. Derfor ble det implementert et JTAG grensesnitt ved hjelp av en FT2232H brikke. Dette tok lang tid å implementere, og det introduserte også flere verktøy, som openOCD, å sette seg inn i. Likevel var det ikke bortkastet tid fordi det gjorde utviklingen videre mye enklere. Dette er også et verktøy som kan tas med videre i utviklingen av operativsystemet.

- Det ble besluttet å utvikle en SD kort driver for operativsystemet. Dette var svært vanskelig og tidkrevende, da det måtte implementeres en EMMC-driver og FAT-driver. Det var likevel helt nødvendig fordi uen funksjonalitet for å laste fra sd-kortet ville det vært umulig å implementere prosesser, fordi programmer måte ha blitt kompilert sammen med kjernen.

- De har vært et ønske å kunne distribuere KybOS over flere noder. Det er ikke blitt implementert fordi dette krever en nettverksmodul som KybOS kan kommunisere over. Det beste alternativet er en ethernet modul slik at Raspberry Pi-en kan kobles til internett. Men det

å lage en ethernet driver ville ha være svært tidkrevende. Derfor ble ikke distribusjons-delen av KybOS implementert, men heller lagt til rette for slik at denne jobben kan gjøres senere.

- I kapittel 14 ble det utført en del tester. Responstesten her var ikke spesielt god for KybOS. Den hadde noen utliggere som var svært dårlig. Dette er katastrofalt for sanntidsegenskapene. Dette må forbedres ved videre utvikling. Årsaken til utliggerne er ikke kjent.

15.4 Konklusjoner

Dette prosjektet har stort sett vært vellykket. Målene beskrevet i 1.2 har i stor grad blitt gjennomført, og en robust og fleksibel kjerne har blitt implementert. Dette dokumentet er beregnet til å veiledet fremtidige utviklere for å forstå både hensikten og tankegangen bak KybOS, samt hvordan KybOS fungerer, slik at de kan utvikle systemet videre.

Testene viser at sanntidsegenskapene ikke er gode nok. Kontekstbyttene burde optimaliseres. Responstesten i 14.2 hadde noen utliggere som var svært dårlig. Kilden til disse må finnes og utbedres. Først da kan KybOS ha gode sanntidsegenskaper.

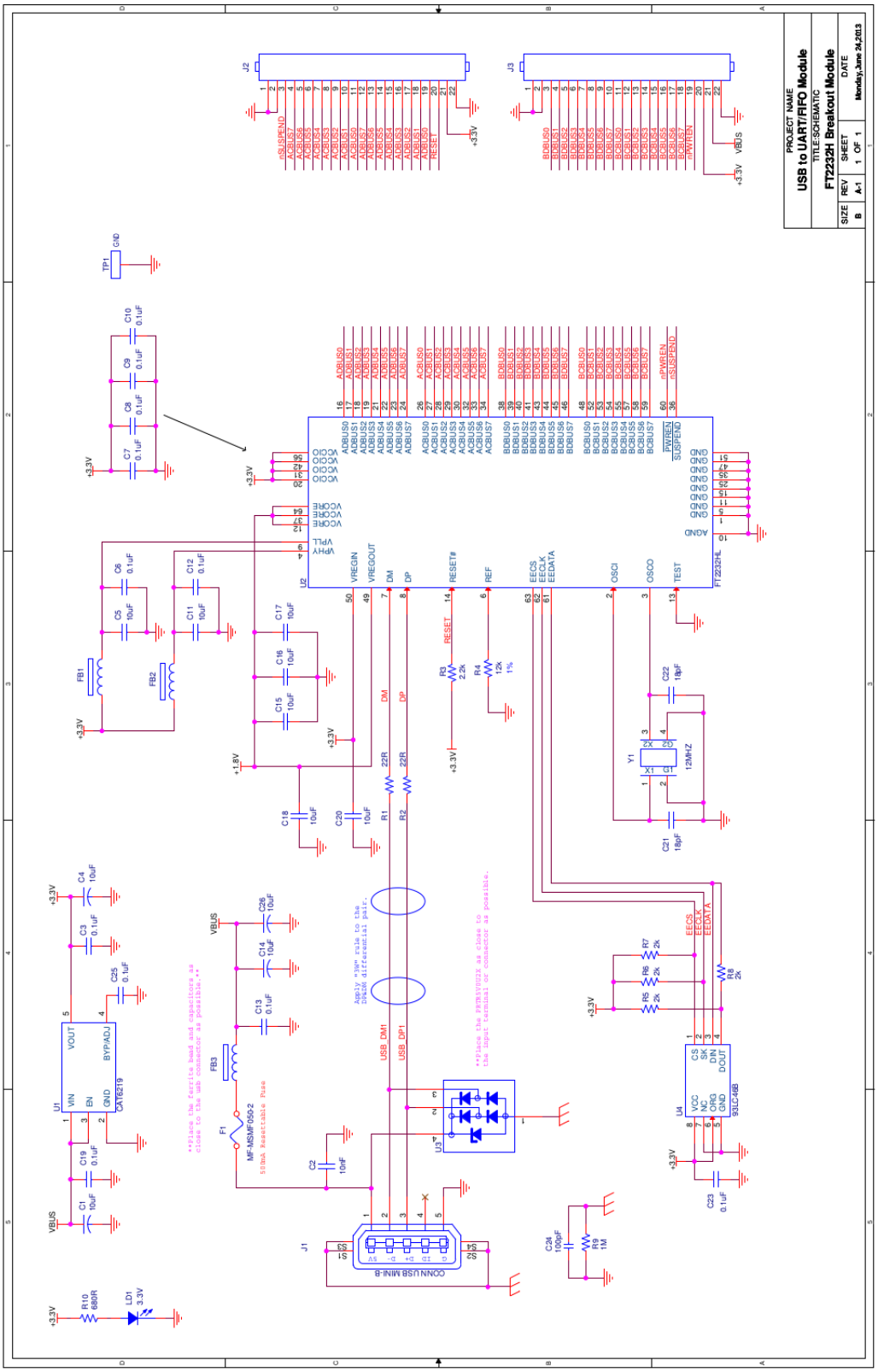
Operativsystemet er imidlertid ikke på noen måte ferdig. Dette er en prototype som kan videreutvikles slik at det kan bli nyttig for utvikling. Det endelige målet er å oppnå et distribuert operativsystem. Dette har ikke blitt implementert i stor grad, men er blitt lagt til rette for ved designet av kjernen.

15.5 Videre arbeid

Hovedfokuset på videre utvikling burde være å tilrettelegg for å distribuere systemet. I tillegg kan sanntidsegenskapene til systemet forbedres ved å optimalisere kontekst byttene og å implementere en bedre planlegger. For eksempel burde en tak-algoritme implementeres for å unngå prioritets invertering. Her er noen hovedpunkter som burde tas tak i for å videreutvikle KybOS:

- Optimalisere kontekst byttene.
- Implementere bedre algoritmer for planleggeren.
- Finne kilden til utliggerne ved responstesten i kapittel 14.2
- Bruke postboks funksjonaliteten slik at man kan kjøre med alle fire prosessorene på chip-en i stedet for bare en.
- Utvikle en ethernet driver.
- Implementere nettverksmodulen som legger til rette for å distribuere operativsystemet.

A Kretskjema av JTAG adapteren



Figur A.1: Kretskort av JTAG adapteren beskrevet i kapittel 5.2

Referanser

[1] Openocd documentation, 2015. URL http://openocd.org/documentation/.

[2] Gcc cross compiler, 2015. URL http://wiki.osdev.org/GCC_Cross-Compiler.

[3] Raspberry pi firmware, 2015. URL www.github.com/raspberrypi/firmware.

[4] Newlib documentation, 2015. URL https://sourceware.org/newlib/.

[5] Executable and linkable format, 2016. URL http://wiki.osdev.org/ELF.

[6] Mailbox property interface, 2016. URL https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface.

[7] ARM. *ARM Architecture Reference Manual: ARMv7-a and ARMv7-R edition*, c.c edition, 2014.

[8] Technical Committee SD Association. *SD Specifications Part A2 SD Host Controller Simplified Specification*, 3.00 edition, 2011.

[9] Technical Committee SD Association. *SD Specifications Part 1 Physical Layer Simplifier Specification*, 4.10 edition, 2013.

[10] Broadcom. *BCM2835 ARM Peripherals*, 2012.

[11] A. Murstad. Sanntidstesting av responstid for operativsystemer på rasberry pi. Technical report, Institutt for Teknisk Kybernetikk, 2015.

[12] E. W. Solnør. Bygge et mikrokjerne og meldingsbasert operativsystem. Technical report, Institutt for Teknisk Kybernetikk, 2015.

[13] W. Stallings. *Operating Systems: Internals and Design Principles*. Pearson, 8th edition, 2014.

[14] A. S. Tanenbaum. *Modern Operating Systems*. Pearson, 3rd edition, 2007.