Lego Mindstorm™ Netværk

DROP™

Gruppe E2-111	Vejleder
Thomas Ledet	Brian Nielsen
Søren Nøhr Christensen	
Mads Wiederholdt Jensen	
Jens Gorm Rye-Andersen	
Dennis Micheelsen	
Kim Enevold Jensen	

Faculty of Engineering and Science
University of Aalborg

Department of Computer Science

TITEL:
Lego Mindstorm Netværk

UNDERTITEL:
DROP - Direct RCX Open Protocol

SEMESTERPERIODE:
DAT2,
1st of February - 30th of May 2002

PROJEKTGRUPPE:
E2-111

GRUPPEMEDLEMMER:
Thomas Ledet, ledet@cs.auc.dk
Søren Nøhr Christensen, snc@cs.auc.dk
Mads Wiederholdt Jensen, blonde@cs.auc.dk
Jens Gorm Rye-Andersen, jgra@cs.auc.dk
Dennis Micheelsen, sak@cs.auc.dk
Kim Enevold Jensen, kej@cs.auc.dk

VEJLEDER:
Brian Nielsen, bnielsen@cs.auc.dk

ANTAL KOPIER: 8

ANTAL SIDER: 94

SYNOPSIS:

Denne rapport beskriver udviklingen af en netværksprotokol til en Lego **RCX** klods, med styresytemet LegOS. Netværket er implementeret ved hjælp af sensor- og motorportene istedet for **IR-porten**.

Analysedokumentet beskriver generel teori omkring operativsystemer og netværk, samt en analyse af LegOS og netværksprotokollen **LNP**. Den generelle teori omkring netværk fokuserer på **OSI-modellen**. Envidere indeholder dokumentet også en gennemgang af RCX'ens hardware.

Designdokumentet beskriver designet af en protokolstak med to lag, det logiske lag og datalink laget. Designet af datalink laget er bygget op omkring en **alternating bit protokol**.

Thomas Ledet	Søren Nøhr Christensen
Mads Wiederholdt Jensen	Jens Gorm Rye-Andersen
Dennis Micheelsen	Kim Enevold Jensen

Indhold

1 Indledning

Indholdsfortegnelse

Denne indledning har til formål at klarlægge vores motivation for at arbejde med netværk på LEGOs **Robotic Command eXplorer**(**RCX**) klodsen. Vi vil kort opsummere på det arbejde, der allerede er gjort med hensyn til operativsystemer og netværk på RCX'en. Derudover vil hardwaren i RCX'en blive beskrevet kort, med hensyn til de helt basale dele. Denne indledende analyse vil resultere i en systemdefinition for projektet.

1.1 Hvad er Lego Mindstorms

Lego Mindstorms er en serie Lego produkter, der er baseret på en **RCX** mikrocomputer (se figur 1.1). Et af produkterne er et **Robotics Invention System** (**RIS**). **RIS** består af omkring 700 legoklodser, to motorer, to tryksensorer, en lyssensor, en infrarød sender, en CD-Rom med software og hjertet i det hele: RCX-mikrocomputeren.

RCX'en er en mikrocomputer baseret på Hitachi H8/300L mikroprocessoren, som er bygget ind i en stor "legoklods". Legoklodsen er udstyret med et **Liquid Crystal Display** (**LCD**, fire knapper og syv I/O porte. De fire knapper bruges til at tænde og

Figur 1.1: RCX klodsen.

slukke RCX'en, skifte program og til at starte og stoppe et program. Der er også en View-knap, der ikke bruges i LegOS. Der er tre inputporte, som bruges til at forbinde sensorerne med RCX'en. De tre output porte bruges til at styre motorerne, og endelig er der en **infrarød port**, der bruges til kommunikation med for eksempel en PC.

Ideen med Lego Mindstorms er, at robotterne bygges af legoklodser, og disse bliver styret af RCX'en. Til at skrive programmerne til RCX'en bruges en PC. Programmerne bliver downloadet til RCX'en ved hjælp af RCX'ens indbyggede IR-port og den medfølgende IR-sender, som er forbundet til PC'en.

1.2 Motivation for netværk på RCX

Vi ønsker at udvikle vores egen netværksprotokol til RCX-klodsen. LegOS, som er det RCX operativsystem, vi benytter os af, indeholder i forvejen en netværksprotokol, men denne baserer sig udelukkende på den infrarøde port. Vi ønsker at lave en ny netværksprotokol.

Formålet med at lave en ny netværksprotokol er, at gøre det muligt at bruge kabler til at forbinde RCX'er, frem for den infrarøde kommunikation. Herudover ønsker vi at opnå forståelse for netværksprotokoller generelt, samt indsigt i operativsystemers virkemåde.

Med projektet ønsker vi også at opnå en viden omkring indlejrede systemer. RCX'en er velegnet til dette, på grund af den relativt simple hardwarearkitektur. Arbejdet med at få RCX'erne til at kommunikere vil give indblik i, hvordan en mikrocomputer virker på det nederste niveau. Endvidere ønsker vi at se, om det kan lade sig gøre at få en højere båndbredde via kabler end via IR-porten.

1.3 Eksisterende RCX projekter

RCX klodsen er meget brugt til uddannelsesformål på grund af den simple hardware, sammenlignet med for eksempel en Pentium 4 platform. Dette er også grunden til, at der eksisterer et meget stort antal igangværende projekter, der bruger RCX'en som udgangspunkt.

RCX'en bruges af studerende verden over, til at tilegne sig viden omkring operativsystemer, indlejrede systemer og så videre. Et af de mere kendte og udbredte projekter er LegOS, som er et operativsystem til RCX'en. LegOS projektet er startet af Marcus L. Noga.

På AAU er der udviklet to operativsystemer til RCX'en. Det drejer sig om henholdsvis ChaOS fra 2001 og nossOS fra 2000. Da vi ønsker at arbejde videre på LegOS, vil vi kun gennemgå dette system i detaljer.

1.4 Systemdefinition

Vi ønsker at fremstille en netværksprotokol til kommunikation mellem RCX'er. Protokollen skal tage udgangspunkt i OSI-modellen, og det skal være muligt at adressere enkelte RCX'er i netværket. RCX'erne skal benytte LegOS som operativsystem. LegOS tilbyder en netværksprotokol (LNP) som understøtter kommunikation over IR-porten.

Vi ønsker at erstatte denne med vores egen protokol, som skal understøtte kommunikation gennem kabler via I/O porte, og vi ønsker at kunne garantere, at pakker når frem og er fejlfri. Vi ønsker endvidere, at protokollen skal implementeres med henblik på kommunikation mellem brugerporgrammer. Dette skal implementeres ved at gensende tabte eller beskadigede pakker. Endvidere ønsker vi også at kunne adressere hver enkelt RCX i et netværk med flere klodser.

Udviklingsarbejdet skal foregå i programmeringssproget C med inline assemblerkode. Udover netværksprotokollen ønskes en form for applikation, der drager nytte af den implementerede netværksprotokol. Vi ønsker også et klart defineret interface, som gør det muligt for fremtidige brugere at drage nytte af vores netværksprotokol.

2 Platformen

Indholdsfortegnelse

Dette kapitel er en gennemgang af de indsamlede oplysninger om problemområdet og en opsummering af de tanker, vi har gjort os omkring forskellige aspekter af analysen.

Før vi kan implementere en netværksprotokol, skal vi have en basis for denne, i form af et operativsystem. Vi vil gennemgå den generelle teori, der er nødvendig for at implementere og forstå et operativsystem. Endvidere vil vi beskrive, hvordan hvert af de generelle teoretiske aspekter er implementeret i LegOS.

Som et grundlag for design og implementering af vores egen netværksprotokol, vil vi gennemgå den generelle teori bag netværk og se på den netværksprotokol, som allerede er implementeret i LegOS. Endvidere bliver hardwaren i RCX-klodsen gennemgået.

Analyse af platformen har til formål at skabe et dokument, der skal være en støtte, når der skal tages beslutninger omkring design og implementationen. Dokumentet skal også fungere som et samlet opslagsværk for teorien bag projektet.

Når vi bruger generelle tekniske termer, vil vi første gang bruge hele udtrykket efterfulgt af en forkortelse. Efterfølgende vil vi, om muligt, bruge forkortelsen.

2.1 RCX'en

Hjertet i Lego Mindstorms systemet er RCX'en, som er en microcomputer baseret på en chip fra Hitachi H8/300 serien. Det er muligt at programmere RCX'en via en PC og den indbyggede infrarøde port. RCX'en er udstyret med seks porte, tre til input og tre til output, et **LCD**, fire knapper og den før omtalte infrarøde port. Inputportene bruges til at koble sensorer til systemet. Dette kan være tryk-, lys- eller temperatursensorer. Outputportene bruges til at styre motorerne.

I det følgende beskrives hardwarearkitekturen, som RCX'en stiller til rådighed for programmøren. De brugte kilder er [Corwna], [Corwnb] og [Ef01].

CPU'en

RCX'en er udstyret med en Hitachi H8/300 CPU, der kører med en hastighed på 16Mhz. Instruktionssættet inkluderer register-register aritmetik, 16-bit addition, 16-bit subtraktion, 8-bit multiplikation og 16-bit/8-bit division. Logiske operationer som AND, OR, XOR og NOT er også inkluderet. Endvidere er der en række bit shift og bit manipuleringsoperationer. CPU'en kan også håndtere exceptions ved hjælp af interrupts. Af understøttede datatyper kan nævnes bits, bytes og words, sidstnævnte er to bytes lang. Ved siden af disse er der også en 4-bit packed **BCD** datatype, som fungerer ved, at de to 4-bits halvdele af byten repræsenterer et decimaltal.

CPU'en har seksten generelle 8-bit registre, som også kan fungere som otte 16-bit registre. Disse registre kan bruges til både data såvel som adresser. Herudover findes der to kontrolregistre, nemlig **Program Counter** (**PC**) og **Condition Code Register** (**CCR**).

PC er et 16-bit register, der indeholder adressen på den næste instruktion, som CPU'en skal udføre. **CCR** er et 8-bit register, som indikerer CPU'ens status. Tabel 2.1 på side 8 viser hvilken funktion, de forskellige bits i **CCR** har. Når de generelle registre bruges til adresser, tilgåes de som de otte 16-bit registre (R0-R7). Når de bruges til data, tilgåes de som otte 16-bit registre (R0-R7), eller som seksten 8-bit registre (R0H-R7H og R0L- R7L). Registret R7 bruges også som stakpointer.

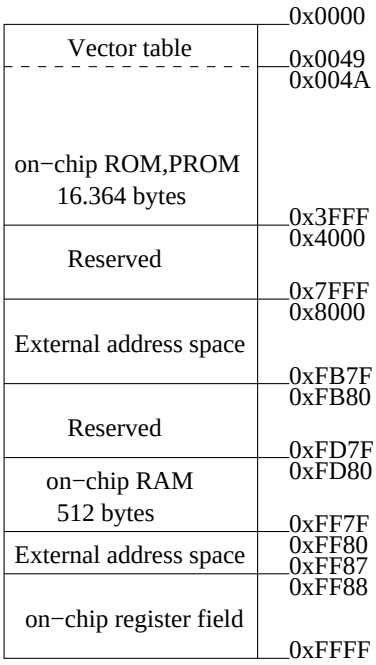
Hukommelse

Hitachi-chippen har som standard 16KB ROM og 512bytes RAM. Udover dette har Lego tilføjet 32KB ekstern RAM. Dele af hukommelsen er reserveret til kernen, resten kan bruges til programmer og anden data. Figur 2.1 på side 8 viser, hvordan

Bit nr.	Name	Beskrivelse
0	Carry flag	
1	Overflow flag	Sættes til 1 hvis et aritmetrisk overflow forekommer
2	Zero flag	Denne bit sættes til 1 hvis en aritmetrisk operation giver resultatet nul
3	Negative flag	Denne bit sættes til 1 hvis en aritmetrisk operation giver et negativt resultat
4	User bit	
5	Half-carry flag	
6	User bit	
7	Interrupt mask	Denne bit bruges til at slå interrupts fra og til

Tabel 2.1: Funktionen af de forskellige bits i CCR

hukommelsen er inddelt på RCX'en. Hukommelsen fra adresse 0x0000 til 0x3FFF



Figur 2.1: Hukommelse på RCX'en

er ROM'en og fra 0x3FFF og op ad ligger RAM'en. De to reserverede områder, der ses på figur 2.1, bruges af RCX'en selv, og må ikke bruges af udvikleren.

Timers

RCX'en har tre indbyggede timere: en 16-bit timer, en 8-bit timer og en såkaldt "watchdog timer". I det følgende beskrives de grundlæggende principper, som timerne er baseret på, frem for at beskrive hver enkelt indgående. Uddybende information kan findes i [Corwnb]

Timere er en essentiel del af ethvert operativsystem. Uden timere på CPU'en, ville det, for eksempel, ikke være muligt at lave tidsinddelt schedulering. I implementeringen af et netværk er timere også en vigtig del, da timere bruges til fejlhåndtering. Dette er beskrevet i detaljer i afsnit 2.3.2 på side 37.

De tre timere bruger en 16-bit **free running counter** (**FRC**) som tidsbase. **FRC**'en er en tæller, der bare tæller op, indtil den nulstilles af programmøren. Den tælles op af en impuls fra enten en intern eller en ekstern "clock source"[1]. I praksis bruges timerne ved at sammenligne deres værdi med en værdi i et af de to **output compare registers** nemlig **OCRA** og **OCRB**. Når de to værdier er ens, genereres et interrupt[Corwnb]. 8-bit timeren er også baseret på den 16-bit **FRC**, men har en 8-bit tidsbase og to kanaler.

A/D konverteren

Analog/Digital konverteren (**A/D konverteren**) laver analoge signaler fra input portene om til digitale signaler som CPU'en kan bearbejde. Som eksempel kan det nævnes, at A/D konverteren skal bruges til at lave analoge signaler fra lyssensorerne om til digitale signaler, som CPU'en kan regne på. Det samme gælder for temperatursensorer. A/D konverteren i RCX'en kan arbejde i to forskellige 'modes', nemlig **Single mode** og **Scan mode**, som begge forklares nedenfor. Resultatet af en konvertering gemmes i fire 16-bit registre (**ADDRA, ADDRB, ADDRC, ADDRD**). Fra disse registre kan CPU'en hente de data som A/D konverteringen har resulteret i. Konverteren er en 10-bit "successive-approximations" konverter og den har otte input kanaler. Med "successive-approximations" menes det, at hver af de otte input kanaler bliver konverteret uden afbrydelser[Ef01].

Single mode konvertering bruges kun, når signalet fra en kanal skal konverteres[Corwnb].

Scan mode kan anvendes, hvis der ønskes konvertering af analoge signaler fra grupper af inputkanaler. Valget mellem de to modes styres af bit 4 i **ADSCR** registret, hvor 0 indikerer **single mode**[Corwnb].

[1] Den tælles enten op på hveranden, hvert 8., eller hver 32.

Serial Communication Interface

Hitachi-chippen indeholder et on-chip **Serial Communication Interface** (**SCI**). Kommunikation med IR-porten foregår gennem dette interface.[Cap01]

Under afsendelse af data, holder **SCI** øje med **Serial Status Register** (**SSR**). Er **TDRE**-bitten i **SSR** sat til 0, ved **SCI**, at **Transmit Data Register** (**TDR**) indeholder nyt data. Dette data kopieres over i **Transmit Shift Register** (**TSR**), hvorefter det sendes afsted over IR-porten. Dette gentages, til der ikke er mere data, og **TDRE**-bitten sættes til 1, hvorefter der kastes et **TSR-empty-interrupt**.[Corwna]

Ved modtagelse af data lytter **SCI** efter en start bit på modtagerlinien. Data flyttes derefter ordnet ind i **Receive Shift Register** (**RSR**). Når der modtages en stop bit, flyttes data fra **RSR** over i **Receive Data Register** (**RDR**), og der kastes et interrupt.

Interrupts på RCX'en[Corwnb]

RCX'en understøtter fire eksterne kilder, nemlig `Non Maskeable Interrupt` (`NMI`) og **IRQ0 - IRQ2**. Der er også 19 interne interruptkilder. Alle interne og eksterne interrupts, undtaget NMI, kan slåes fra ved hjælp af Interrupt Mask bit'en i CCR registret. NMI er den interrupt, der har højeste prioritet. Hvis mere end een interrupt bliver anmodet på samme til, vil den med højeste prioritet blive behandlet først.

Tabel 2.2 fra [Ef01] viser de interrupts, der er tilgængelige på RCX'en.

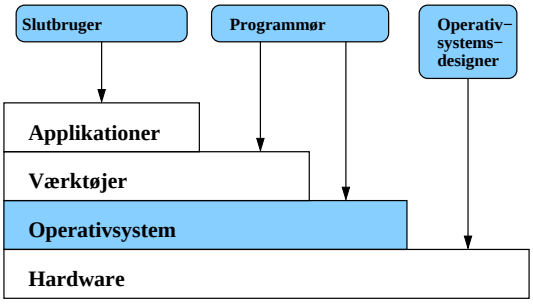
Name	ROM vec.	RAM vec.	ROM handl.	Description
NMI (*)	0x0006	0xFD92	None	Non maskable Interrupt
IRQ0 (*)	0x0008	0xFD94	0x1AB8	Interrupt 0 (run button)
IRQ1 (*)	0x000A	0xFD96	0x294A	Interrupt 1 (on/off button)
IRQ2 (*)	0x000C	0xFD98	None	Interrupt 2
ICIA	0x0018	0xFD9A	None	16-bit timer, input capture A
ICIB	0x001A	0xFD9C	None	16-bit timer, input capture B
ICIC	0x001C	0xFD9E	None	16-bit timer, input capture C
ICID	0x001E	0xFDA0	None	16-bit timer, input capture D
OCIA	0x0020	0xFDA2	0x36BA	16-bit timer, output compare A
OCIB	0x0022	0xFDA4	None	16-bit timer, output compare B
FOVI	0x0024	0xFDA6	None	16-bit timer, overflow
CMI0A	0x0026	0xFDA8	None	8-bit timer 0, compare match A
CMI0B	0x0028	0xFDAa	None	8-bit timer 0, compare match B
OVI0	0x002A	0xFDAc	None	8-bit timer 0, overflow
CMI1A	0x002C	0xFDAE	None	8-bit timer 1, compare match A
CMI1B	0x002E	0xFDB0	None	8-bit timer 1, compare match B
OVI1	0x0030	0xFDB2	None	8-bit timer 1, overflow
ER1	0x0036	0xFDB4	0x30A4	Serial receive error
RXI	0x0038	0xFDB6	0x2C10	Serial receive end
TXI	0x003A	0xFDB8	0x2A9C	Serial TDR empty
TEI	0x003C	0xFDBA	0x2A84	Serial TSR empty
ADI	0x0046	0xFDBC	0x3B74	A/D conversion end
WOVF	0x0048	0xFDBE	None	Watchdog timer overflow

Tabel 2.2: Interrupts på RCX'en. Eksterne interrupts er markeret med (*) resten er interne. Tabellen er sorteret, så den interrupt med højeste prioritet er øverst[Corwnb].

2.2 Operativsystemer

I dette afsnit beskrives de basale dele af et **operativsystem** (**OS**), eller styresystem, som er de to ord, der vil blive brugt. Endvidere vil afsnittet beskrive, hvordan de forskellige funktionaliteter er implementerede i LegOS. Afsnittet skal fungere som opslagsværk omkring LegOS og operativsystemer generelt.

Et operativsystem er software, der har til opgave at kontrollere afviklingen af brugerprogrammer, og skabe et interface mellem hardwarelaget og applikationslaget (figur 2.2). Når et brugerprogram skal bruge en hardwareenhed, skal det ikke selv implementere adgangen dertil. Det sørger OS'et for med en række instruktioner, hvor håndteringen af disse enheder er implementeret. Brugerprogrammet skal derfor bare udnytte det interface , som OS'et tilbyder. Operativsystemet kører som



Figur 2.2: Lagene på en computer med et OS, værktøj og program

en proces på CPU'en, som et almindeligt program ville. Det skal administrere og afvikle andre programmer på systemet, og tildele dem de ressourcer, de har brug for. Derfor er det nødvendigt for operativsystemet at slippe CPU'en og lade andre processer få adgang. Men CPU'en giver ikke adgang til diverse eksterne enheder og porte. Adgangen til disse enheder implementeres ved hjælp af device drivers.

OS'et skal også styre hukommelsesallokeringen til diverse kørende programmer. Dette gøres ved hjælp af en **memory manager**. **Memory manageren** kontrollerer, hvilke områder af hukommelsen, der er i brug. Og den tildeler hukommelsesområder til brugerprogrammer, når de beder om det.

En anden vigtig ting i et OS, er proceshåndteringen. Multitasking, altså det at kunne afvikle flere processer på samme tid, er nødvendigt i moderne OS'er. Det skal siges, at der skelnes mellem samtidig afvikling og parallel afvikling. I et enkeltprocessorsystem, er der ikke noget, der hedder ægte beregningsparallellitet. Derfor skal OS'et styre, hvilken proces, der på et givet tidspunkt skal have adgang til CPU'en.

OS'et skal også kunne håndtere ekstern kommunikation til andre klodser eller en

PC. Dette kunne for eksempel være i form af en infrarød port. Der skal være et interface til en sådan port. Det implementeres med en driver og en protokol, som håndterer kommunikationen over porten.

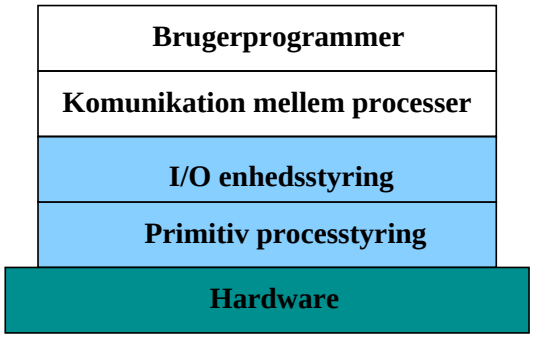
I de følgende afsnit vil de vigtigste funktioner i et OS blive gennemgået. Vi vil forsøge at holde os til de dele, der er relevante for et OS til en RCX. Vi vil også forklare, hvordan LegOS implementerer disse funktioner. De vigtigste kilder for afsnittene er [Sta01], [Ef01] og [Nie00].

2.2.1 Kernestruktur

Opbygningen af kernestrukturen kan følge to principper: monolitisk kernearkitektur og mikrokernearkitektur.

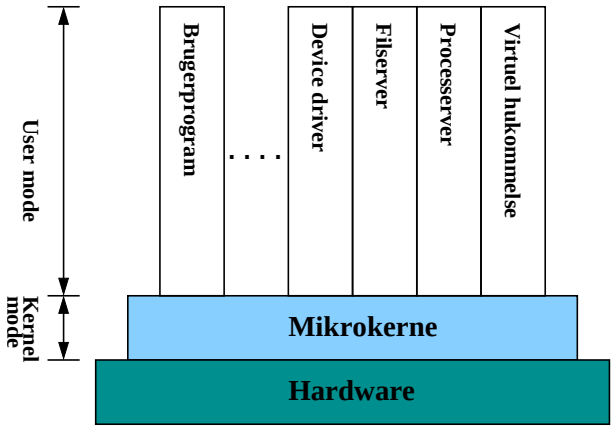
Et OS tager sig af al kommunikation mellem hardware og de brugerprogrammer, som nu engang kører på maskinen. Kernen bruges til processtyring og I/O-enhedshåndtering. Ydermere er den altid resident i hukommelsen.

Den monolitiske kerne har indtil nu været den dominerende kernearkitektur indenfor OS'er. Idéen er at samle OS-instruktioner i selve kernen - det vil sige hukommelsesstyring, filadgang, processorschedulering, drivere og lignende. Ofte vil en monolitisk kerne være implementeret som en enkelt proces. se figur 2.3.



Figur 2.3: Monolitisk kerne

Mikrokernen har den filosofi, at kernen opbygges med kun de vigtigste OS instruktioner. Er der brug for andre instruktioner, end dem, der ligger i kernen, skal de implementeres som moduler. Hvert modul vil køre som en seperat proces, der af kernen styres som en regulær brugerproces. se fig. 2.4. **User mode** er det mode som



Figur 2.4: Mikrokerne

brugerens programmer kører i. Det er et mere begrænset mode end **kernel mode**, som kernen opererer i. **Kernel mode** er kun beregnet til OS specifikke funktioner.

Kernestrukturen i LegOS

Kernen i LegOS er **monolitisk**. Det viser sig ved, at alt kildekoden bliver kompileret sammen i en binær fil, og derved kører hele kernen altid. Kernen kan startes i enten **single tasking** eller **multitasking mode**. Hvis ikke **task manageren** er kompileret med i kernen, er kun **single tasking mode** tilgængeligt.

Når kerneprocessen bliver startet, startes tre processer sammen med den. Den første er en **idle** proces, som har den laveste prioritet, og kører, når ingen andre processer efterspørger CPU-tid. De to andre processer bruges til at implementere dynamisk downloading af brugerprogram. Disse processer hedder henholdsvis **packet_consumer** og **key_handler**. Den første, **packet_consumer**, styrer aktivitet på IR-porten, og bruges til at hente brugerprogrammer ned på RCX'en. Nummer to, **key_handler**, bruges til at håndtere brug af knapperne på RCX'en.

Herefter startes **task manageren**, og den begynder at skifte mellem afviklingen af de tre processer.

Konklusion

Mikrokernens modulopbyggede struktur gør den utrolig fleksibel, og endvidere er det muligt at konfigurere den til specifikke applikations- eller systemkrav. Men en

monolitisk kerne er betydeligt hurtigere på **runtime**[2], da mikrokernens moduler kører i hver sin proces, hvilket skaber en hel del processkifte, som tager tid. Der skal med andre ord afgøres, hvad der er vigtigst for OS'et. Dog er det muligt, at lave et kompromis mellem de to kernestrukturer. Hvis det først gøres klart, hvilke instruktioner, der oftest vil blive brugt, og hvilke der er beregningskrævende, kan disse instruktioner placeres i selve kernen og ikke i et modul.

2.2.2 Hukommelsesstyring

I dette afsnit vil vi diskutere forskellige strategier for hukommelsesstyring, og hvordan LegOS håndterer dette.

Hukommelsen er delt op i to dele: en del til OS'et og en del til brugerprogrammer. Delen til brugerprogrammer skal endvidere deles op så hver proces har sin egen plads i hukommelsen. Vi vil komme ind på tre forskellige metoder til styring af hukommelsen: **fixed partitioning**, **dynamisk partitioning** og **buddy-systemet**. Metoder for **paging** og **segmentation** vil ikke blive beskrevet. Disse metoder er ikke relevante, i det hardwaren i RCX'en ikke understøtter dem.

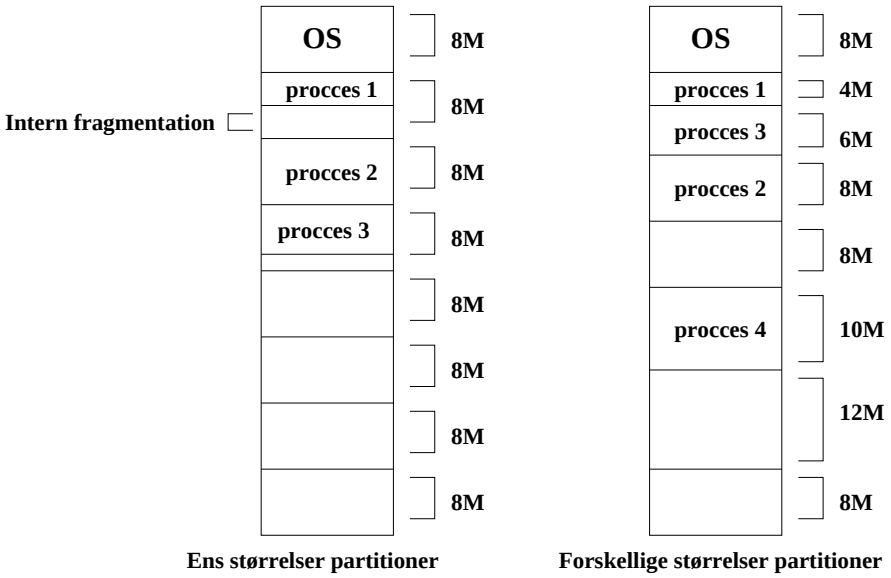
Fixed partitioning deler hukommelsen op i et antal faste partitioner. Partitionerne kan være af samme størrelse, eller de kan være af stigende størrelse. En proces kan derved gemmes i en partition af samme eller større størrelse. Gemmes processen i en større partition, fremkommer der såkaldt intern fragmentation. Det vil sige, at resten af en partition ikke kan bruges. En anden ulempe er, at antallet af aktive processer er begrænset til antallet af partitioner (se figur 2.5 på side 16).

Dynamisk partitioning opretter partitioner dynamisk på runtime. Hver partition er nøjagtig lige så stor som processen, og derved er der ingen intern fragmentering. Men efterhånden vil hukommelsen alligevel blive fragmenteret, dog ved ekstern fragmentering. Ekstern fragmentering er de små blokke af hukommelse, som opstår mellem partitionerne (se figur 2.6 på side 17).

Ekstern fragmentering gør det nødvendigt med **compaction**, hvilket er en teknik, som fjerner den eksterne fragmentering, ved at samle den frie hukommelse i én blok. Ulempen er bare, at **compaction** kræver en del CPU-tid, og systemet skal understøtte dynamisk relokation. Det vil sige, at programmer, som ligger i hukommelsen kan håndtere, at de bliver flyttet rundt i hukommelsen.

Der findes forskellige placeringsalgoritmer til dynamisk partitioning.

[2] Der skelnes mellem compile-time og runtime



Figur 2.5: Fixed partitioning

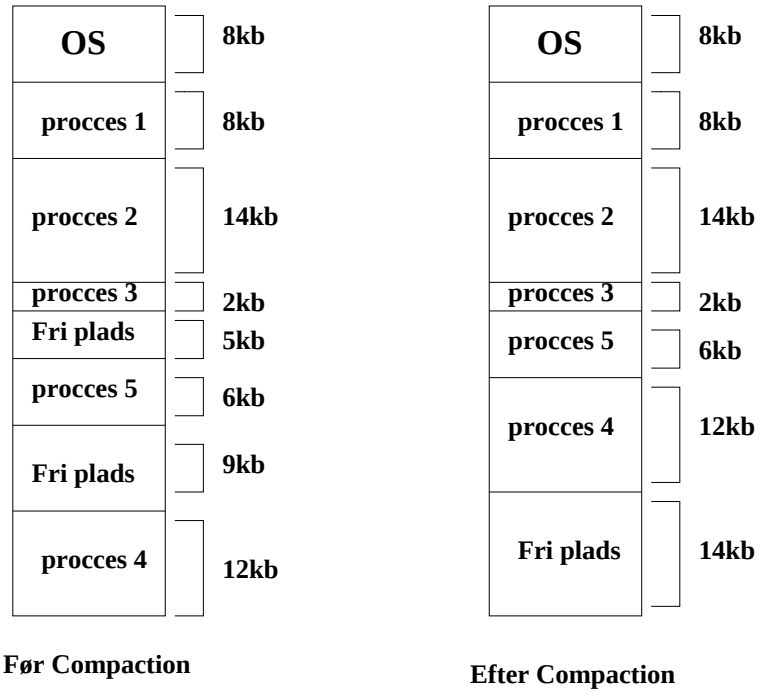
first-fit algoritmen leder igennem hukommelsen, indtil den finder den første ledige blok med tilstrækkelig størrelse. Det giver en hurtig allokering, men på bekostning af en høj fragmentering.

next-fit algoritmen opretholder en pointer til det sidste sted, den har placeret en proces. Når den skal indsætte en ny, laver den det, der svarer til en **first-fit** med pointeren som start sted. **Next-fit** har stort set de samme fordele og ulemper som first-fit. En af ulemperne er, at den største portion fri hukommelse, som normalt er i slutningen af hukommelsen, hurtigt bliver fragmenteret, i forhold til **first-fit**, som spreder fragmenteringen ud over hele hukommelsen.

best-fit algoritmen leder hele hukommelsen igennem for at finde den ledige blok, som passer bedst i forhold til den nye proces. På trods af navnet er best-fit sjældent det bedste valg. Den er langsom, da den leder hele hukommelsen igennem, og samtidig skaber den hurtigt fragmentering med meget små ledige blokke.

worst-fit algoritmen placerer altid processer i den største ledige hukommelsesblok. Selv om navnet antyder noget andet, er denne algoritme ofte det bedste valg[3]. Da den altid vælger den største ledige blok, vil den ikke skabe helt så mange små fragmenter. Dog er den ikke så hurtig som first-fit og next-fit.

[3] Se [Sta01] kap. 7



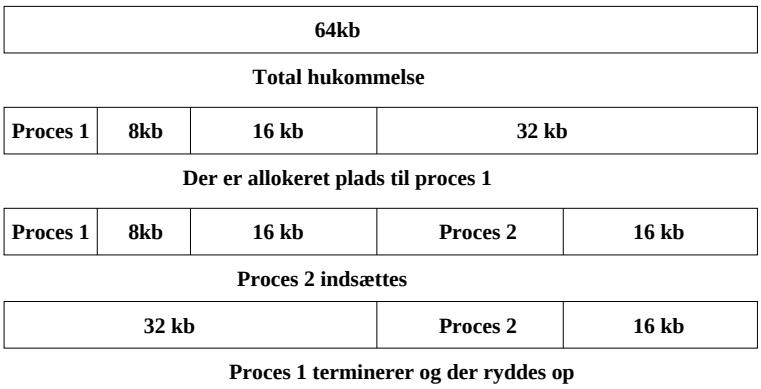
Figur 2.6: Dynamisk partitioning

Buddy systemet betragter hele den frie hukommelse som én blok. Når der er brug for at allokere plads til en proces med størrelsen s, deles blokken op i to lige store dele. Dette gentages, indtil der er en blok P, med størrelsen p, hvor det gælder, at $p/2 < s \leq p$. Når en blok deles i to halvdele, kaldes de to for buddies. Når **memory manageren** frigør hukommelse, bliver to tomme buddies lagt sammen til én blok. Figur 2.7 viser hvordan **Buddy-systemet** opdeler hukommelsen.

Med buddy systemet er det ikke nødvendigt med **compaction**, da det selv rydder op efter sig. Dog er det ikke så simpelt at implementere som de andre algoritmer.

Hukommelsesstyring i LegOS

LegOS bruger dynamisk partitioning implementeret ved en simpel **first fit** algoritme til at allokere hukommelse til brugerprocesserne. I det følgende vil vi kigge på de funktioner, der er til rådighed i *kernel/mm.c* og som håndterer hukommelsesstyring. RCX'ens hardware understøtter ikke nogen form for avanceret hukommelsesstyring, som **paging** eller **segmentation**. Valget af en simpel allokeringsalgoritme kan tilskrives den meget begrænsede mængde hukommelse, som er til rådighed på RCX'en.



Figur 2.7: Opdelingen af hukommelsen med **Buddy-systemet**

En mere avanceret hukommelsesallokering ville formentlig skabe for meget overhead, til at det kunne betale sig. LegOS deler hukommelsen op i to dele, en kernedel og en del til brugerprogrammer. Kernedelen ligger i adresserummet fra 0x8000 til `mm_start`. Hukommelsen fra `mm_start` til 0xFFFF står til rådighed for **memory manageren**, se figur 2.1.

For at holde styr på hukommelsen bliver den delt ind i blokke. I hver hukommelsesblok bliver de to første bytes brugt til at holde en **process identifikation** (**PID**), der fortæller hvilken proces, der har bedt om at få pågældende blok allokeret. Hvis blokken ikke er allokeret, er værdien i disse bytes `MM_FREE`. De næste to bytes i blokken bruges til at gemme størrelsen på blokken, som skal være et lige antal bytes. Figur 2.8 på side 18 viser de blokke, som bliver allokeret under opstart af kernen. Til at allokere hukommelse til kerne eller bruger-

&mm_start FREE `0xEF30` LCD DATA
`0xef50` FREE
`0xf000` Motor `0xF010` FREE `0xFB80` Vectors `0xFE00` FREE `0xFF00` OnChip Register

Figur 2.8: Hukommelsesallokeringen som den ser ud efter opstart af kernen.

processer bruges funktionen `malloc()`. Det første `malloc()` gør, er at se om semaforen `mm_semaphore` er taget, for at sikre, at ingen andre processer manipulerer hukommelsen. Hvis `malloc()` får adgang til den region, som den spørger om, vil den begynde at lede efter en blok hukommelse med en passende størrelse.

Pointeren `mm_first_free` peger på den første ledige hukommelsesblok, og det er her `malloc()` starter med at lede. Hvis ikke `malloc()` finder en brugbar hukommelsesblok, returnerer den `null`.

Til at frigive hukommelse bruges `free()` funktionen, som frigør dynamisk allokeret hukommelse. Hvis frigørelse skal ske efter at en proces er termineret, bruger **memory manageren** funktionen `mm_reaper()`, der benytter sig af en algoritme, der minder meget om en **Mark and Sweep** algoritme[4]. Funktionen `mm_reaper()` løber igennem alle hukommelsesblokkene to gange. I første gennemløb markerer den alle de blokke, som tilhører den proces, der terminerer. Kerneprocesser markeres ikke. I andet gennemløb frigør `mm_reaper()` de hukommelsesblokke, som er markeret.

LegOS' **memory manager** rydder op efter ekstern fragmentering. Funktionen `mm_try_join()` tager adressen på en fri hukommelsesblok som parameter. Funktionen søger fremad i hukommelsen fra den adresse, og samler alle fri hukommelsesblokke. Dette gør den indtil den møder en blok, der ikke er fri.

Konklusion

Denne måde LegOS styrer hukommelsen på, giver ikke nogen form for beskyttelse af hukommelsen, ligesom hardwaren heller ikke beskytter hukommelsen. Dette bevirker, at brugerprogrammer kan skrive overalt i RAM'en, og eventuelt overskrive regioner, hvor OS'et ligger. Endvidere kan brugerprogrammer dynamisk allokere mere hukommelse, hvilket muliggør at systemet løber tør for RAM. Konklusionen på dette er, at brugerprogrammer skal skrives med forsigtighed, for at undgå at systemet går ned.

2.2.3 Processtyring

En af de vigtigste opgaver for et OS er processtyringen. Hver proces skal tildeles ressourcer (hukommelse, processortid og lignende). Det skal være muligt for processer at dele og udveksle informationer. Ressourcerne for hver proces skal ydermere beskyttes mod andre processer, og det skal være muligt med synkronisering mellem processer. For at overholde disse krav, er det nødvendigt at opretholde en datastruktur for hver proces, som beskriver tilstanden og ejerskabet af processen. I det følgende vil vi gennemgå forskellige aspekter af processtyring.

[4] **Mark and Sweep** er en algoritme til garbage collection og er beskrevet i [Nor02]

Procestilstande - På et multiprogramming uniprocessor system, hvor der kører flere processer på samme tid, vil processerne nødvendigvis befinde sig i forskellige tilstande. Disse tilstande skal beskrives af OS'et, og de bruges i processtyringen.

I den simpleste model, kaldet **Two-state model**, kan processerne befinde sig i to tilstande: **RUNNING** eller **NOT RUNNING**. Det er nødvendigt med en datastruktur til at holde de processer, som befinder sig i tilstanden **NOT RUNNING**. Kun den proces, som kører på processoren lige nu er **RUNNING**. Processer, der er **NOT RUNNING**, gemmes i en kø-struktur, hvor de skiftevis får adgang til processoren. Men hvis en proces er blokeret, det vil sige venter på, at en hændelse sker, det kunne være at et I/O kald returnerer, skal den ikke have adgang til processoren. Derfor er det nødvendigt med en ny procestilstand. Vi splitter derfor **NOT RUNNING** op i to tilstande: **READY** og **BLOCKED**. Ofte tilføjes yderligere to tilstande til modellen, nemlig **NEW** og **EXIT**.

Denne model er kendt under navnet **Five-state model**, og de fem tilstande er defineret således:

Running - Processen, der kører på cpu'en lige nu.

Ready - En proces, der er klar til at blive afviklet af CPU'en.

Blocked - En proces, der ikke er klar til at blive afviklet, for eksempel fordi den venter på svar fra I/O enheder.

New - En proces, der lige er blevet oprettet, men endnu ikke har fået adgang til mængden af eksekverbare processer.

Exit - En proces, der af OS'et er fjernet fra mængden af eksekverbare processer, enten fordi den er stoppet, eller fordi den har termineret.

Procesbeskrivelse for at holde styr på de forskellige processer, skal OS'et opretholde en såkaldt procestabel. I procestabellen gemmes en oversigt over proces images for hver proces. Et proces image beskriver en proces ved hjælp af følgende punkter:

Bruger Data - Indeholder program data, en brugerstak og programmer, der kan blive modificeret.

Bruger Program - Programmet, der skal afvikles.

System Stak - Til hver proces hører der en eller flere LIFO[5] systemstakke. Stakken bruges til at gemme parametre og adresser for procedurer og systemkald.

[5] Last in, First out

Proceskontrolblok(**PKB**) - Data nødvendig for OS'et til at kontrollere processen.

PKB indeholder en stor mængde data. Generelt set kan dette data deles op i tre dele:

- Procesidentifikation
- Procestilstandsinformation
- Proceskontrolinformation

Procesidentifikation er en unik numerisk værdi tildelt til enhver proces. Den kan bruges som indeks i procestabellen. **Procestilstandsinformation** indeholder en kopi af indholdet af de registre som processen gjorde brug af, da den havde CPU-tid. Når en proces kører, ligger informationen i registrene, men når processen ikke kører, skal den information gemmes til næste gang, processen skal køres. Disse registre indbefatter brugerregistre, kontrol- og statusregistre, samt stakpointere. **Proceskontrolinformation** er en stor del af **proceskontrolblokken**. Her gemmes procestilstanden, prioriteten, afviklingsinformation, samt ID på en eventuel blokerende proces. Derudover er der informationer om **Interprocess Communication** (**IPC**), procesprivilegier, hukommelsesstyring og ressourceejerskab og benyttelse.

PKB'en er den vigtigste datastruktur i et OS. Den indeholder al den information OS'et har brug for om processen. PKB'en læses og modificeres af næsten samtlige af operativsystemets moduler. Praktisk set definerer PKB'en operativsystemets tilstand. Dette skaber uheldigvis det problem, at en fejl i en enkelt rutine kan medføre, at operativsystemet ikke kan styre den pågældende proces.

Proceskontrol - Den vigtigste del af proceskontrollen er muligheden for processkifte. Men først skal processerne oprettes, på følgende måde:

1. Tildel et unikt nummer til den nye proces.
2. Tildel processen det hukommelse den skal bruge.
3. Initialiser **PKB**.
4. Opret de nødvendige pointere.
5. Opret eller udvid andre datastrukturer.

Et processkifte kan kun lade sig gøre på et tidspunkt, hvor OS'et har overtaget kontrollen fra en kørende proces. Dette kan ske ved et **system interrupt**. Det kan

for eksempel være en ekstern I/O enhed, der har færdiggjort sin opgave. Ved et interrupt overfører processoren kontrollen til en **interrupt handler**, som står for at kalde de funktioner som skal håndtere brugen af I/O enheden. Det kan være et af følgende to interrupt typer:

Clock interrupt - OS'et undersøger om den kørende proces har kørt i den maksimalt tilladte tidsperiode. Hvis den har, bliver processen sat i ready-køen og en ny proces køres.

I/O interrupt - Det afgøres hvilken I/O enhed interruptet stammer fra, og processer i **blocked** tilstanden, der venter på denne I/O enhed, overføres af scheduleren til **ready**-køen. Understøtter systemet prioriterede processer, beslutter scheduleren om den kørende proces skal stoppes til fordel for en proces med prioritet.

Et processkifte kan også ske på grund af en **trap**. En **trap** fremkommer ved at der sker en fejl i den kørende proces. Når en **trap** indtræffer, bestemmer OS'et om fejlen er en kritisk fejl. Hvis dette er tilfældet bliver processen termineret og en anden process får CPU tid. Hvis fejlen ikke er kritisk vil OS'et, udfra dets design og fejlens art, udføre en bestemt handling. Det kunne være at forsøge at genskabe processen eller simpelthen at give brugeren besked. Den kan så enten skifte process eller blive ved den nuværende process.

Interrupts er en måde, hvorpå andre enheder kan afbryde processorens normale kørsel. Med andre enheder menes I/O-enheder, brugerprogrammer, timere etc. Tabel 2.3, som er oversat fra [Sta01], viser forskellige klasser af **interrupts**.

Brugen af **interrupts** er en måde at forbedre udnyttelsen af processoren på. I/O enheder er meget langsommere end processoren. Uden **interrupts** ville processoren være tvunget til at vente på, at en I/O-enhed færdiggører det arbejde, den er sat i gang med. Overvej følgende forenklede eksempel:

En computer sender data til en printer, printeren er imidlertid meget langsommere end processoren. Computeren må derfor sende en bid data, og så vente på, at printeren er klar til den næste bid. Den tid, som processoren venter på printeren, er spild af værdifuld processortid. Med indførelsen af **interrupts** kan processoren sende en bid data til printeren og derefter give sig til at håndtere en anden opgave. Idéen er, at når printeren er klar til at modtage ny data, vil den sende et **interrupt** til processoren. Dermed signalerer den til processoren, at "jeg er klar til mere data".

Program interrupt	Genereret af en tilstand, der fremkommer som et resultat af udførelsen af en instruktion, som aritmetrisk overflow, division med nul, forsøg på at udføre en ulovlig maskininstruktion og forsøg på at tilgå hukommelse uden for det tilladte adresserum.
Timer interrupt	Genereret af en timer inde i processoren. Dette giver operativsystemet mulighed for at foretage operationer,der baserer sig på et tidsinterval.
I/O interrupt	Genereret af en I/O-controller, for at signalere at et program kan afsluttes normalt eller for at signalere en række fejltilstande.
Hardware fejl	Genereret af en fejl, som strømsvigt eller en hukommelsesparitetsfejl.

Tabel 2.3: Klasser af interrupts

Processoren kan nu sende nyt data til printeren og derefter vende tilbage til andre opgaver.

Ovenstående eksempel er ret forenklet, men det beskriver grundprincippet omkring **interrupts**. **Interrupts** er en måde, hvorpå der kan opnås bedre udnyttelse af processoren. For mere information om **interrupts**, se kapitel 1.4 i [Sta01]

Procesafvikling - Formålet med procesafvikling er at tildele processortid til de forskellige processer, så systemet kører mest optimalt. Afviklingen kan inddeles i tre forskellige kategorier: **long**, **medium** og **short-term**.

Long-term scheduling beslutter hvilke processer, der skal indsættes i puljen af processer, som venter på at blive eksekveret.

medium-term scheduling beslutter om antallet af processer, der er delvist eller helt i den primær hukommelse, er optimalt.

Short-term scheduling beslutter hvilken af de eksekverbare processer, der skal afvikles.

Long-term scheduling har til opgave at give processerne videre til **short-term scheduling**. Dette princip benyttes ofte i store systemer, hvor lavere prioriterede processer kan holdes tilbage, for at give højere prioriterede processer fornuftig CPU-tid.

Processtyring i LegOS

I det følgende vil vi gennemgå, hvordan processtyringen fungerer i LegOS. Der vil blive kigget på brugerprogrammer, hvordan en proces oprettes, og hvilken datastruktur, der benyttes til at gemme informationer om de enkelte processer, samt hvordan afviklingen foregår.

Processerne i LegOS benytter en liste med et prioriteringssystem med 20 muligheder. Hver af disse prioriteringsniveauer indeholder en dobbelt lænket liste. Denne opbygning gør, at det kun er hukommelsen, der sætter grænser for antallet af processer på et givet niveau. Disse dobbelt lænkede lister indeholder elementer af typen `pdata_t`. Denne datastruktur er **PKB**'en i LegOS. Denne indeholder følgende data:

size_t *sp_save - pointer til stakpointeren

pstate_t pstate - procestilstanden

pflags_t pflags - processens flag

pchain_t *priority - pointer til prioritets nivauet

struct_pdata_t *next - pointer til næste element

struct_pdata_t *prev - pointer til det forrige element

struct_pdata_t *parent - pointer til processens forælder

size_t *stackbase - pointer til starten af stakken

wakeup_t (*wakeup)(wakeup_t) - pointer til den funktion, som skal køres, når en proces returnerer fra **waiting** tilstanden.

wakeup_t wakeup_data - det data, der skal gives med funktionen ovenfor

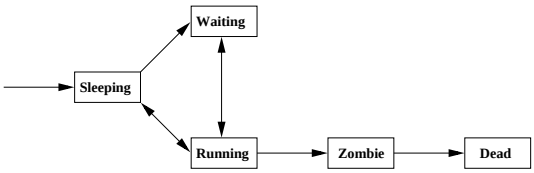
Hvert prioriteringsniveau har en pointer til den nuværende proces, som enten kører, eller lige har kørt.

En proces har 5 tilstande den kan befinde sig i:

- **Dead** - processen har kørt, og det anvendte hukommelse er frigivet fra stakken.
- **Zombie** - processen har kørt, men det anvendte hukommelse ligger stadig på stakken.

- **Waiting** - processen venter på en handling, f.eks et **interrupt**.
- **Sleeping** - processen er klar til at køre.
- **Running** - processen kører

Figur 2.9 viser de fem procestilstande i LegOS. En proces vil ændre tilstand fra **Running** til **Waiting**, hvis den tråd der eksekveres i den, når til et punkt hvor den ikke kan fortsætte, før en given hændelse indtræffer, f.eks et **interrupt** fra en I/O enhed.



Figur 2.9: Procestilstands diagram for LegOS

Når der skal oprettes nye processer bruges `execi()`, som tager en pointer til funktionen, der skal køres, antallet af parametre, de aktuelle parametre, prioriteten, og hvor stor processens stak skal være. `execi()` forsøger at oprette en `pdata_t` i prioritetskøen og hukommelse til funktionens stak. Den nye proces kommer ind i køen af processer i **sleeping** tilstanden.

Procesafviklingen i LegOS benytter **short term scheduling**, i form af en prioriteret **Round-Robin** algoritme. Selve afviklingen foregår på den måde, at når processens tid er udløbet, eller hvis processen frivilligt frigiver CPU'en, bliver scheduleren kaldt. Den beslutter, hvilken proces, der skal køres næste gang. Dette foregår, ved at scheduleren kalder `trywait()`, som kontrollerer **task semaforen**[6]. Hvis denne er blokeret, vil der ikke blive kørt en ny proces. Er **task semaforen** ikke blokeret, vil afvikleren blokere den, og undersøge, hvilken tilstand den aktuelle proces er i. Hvis processen er i tilstanden **zombie**, deallokerer scheduleren processens hukommelse, og fjerner den fra prioritetskøen. Hvis denne proces er den sidste på dette prioritets niveau, fjernes niveauet fra niveaulisten. Herefter vil scheduleren lede efter den proces der skal køres, ved at kigge på næste niveau i prioritets listen. Hvis processen, som den undersøger, har tilstanden **sleeping**, er processen klare til at blive afviklet, og dens tilstand bliver sat til **running**. Hvis en proces er i tilstanden **waiting**, undersøges den næste proces i rækken. Hvis der ikke findes nogen egnede processer fortsættes til næste niveau.

[6] En semafor, som sørger for, at kun en proces afvikles ad gangen

Konklusion

Problemet med sammensætning af delte beskyttede ressourcer og en prioriteret liste i processtyringen, kan give problemer. Hvis en proces med lav prioritet låser ressourcer, som en proces med højere prioritet ønsker at benytte, vi den lavere prioriterede proces blokere for afviklingen af den højere prioriterede proces. Der er i LegOS ikke gjort noget for at forhindre dette[Nie00].

2.2.4 Samtidighed

Når en række processer skal køre samtidigt på en processor, og have adgang til delte ressourcer, opstår der en række problemer. Dette er problemer omkring gensidig udelukkelse, udsultning, deadlock og livelock. Følgende afsnit handler om disse problemer.

Gensidig udelukkelse Når flere processer skal deles om den samme fælles ressource, uden at tilgå den på samme tid, opstår kravet om gensidig udelukkelse. En ressource kaldes en kritisk ressource, hvis det ovenstående gælder, og den bestemte del af processen, som tilgår den kritiske ressource kaldes for den kritiske region. Implementeringen af gensidig udelukkelse fører til de tre andre problemer omkring samtidig afvikling af processer.

Udsultning er et problem der opstår, hvis en eller flere processer aldrig får adgang til en delt ressource. Dette kan for eksempel ske, hvis en række processer med højeste prioritet skiftes til at tilgå en ressource på en måde, således at en proces med lavere prioritet aldrig vil få adgang til ressourcen.

Deadlock kan forekomme, hvis to kritiske regioner afhænger af den samme kritiske ressource. Eksempelvis ville **deadlock** opstå i det følgende eksempel.

Der er to processer, nemlig **P1** og **P2**, og der er to ressourcer **R1** og **R2**. **P1** har taget kontrollen med **R1** og **P2** har taget kontrollen med **R2**. Hvis begge processer skal bruge den anden ressource for at komme videre, opstår der deadlock. Begge processer går ind i en evig indbyrdes venten på hinanden.

Livelock forskellen fra **deadlock** til **livelock** er, at der findes en eksekveringssekvens, hvor processerne ikke vil blokere hinanden. Altså vil **livelock** kun opstå hvis for eksempel to processer gang på gang prøver at tilgå den kritiske ressource

på præcis samme tidspunkt. Hvis den ene proces' udførelseshastighed i dette tilfælde varierer lidt fra den anden proces vil en af dem kunne tilgå den kritiske ressource og de ville dermed ryge ud af deres **livelock**.

Semaforer er en mekanisme der kan anvendes til at sikre, at en ressource kun tilgåes af en proces ad gangen. **Semaforer** består af en **semaforværdi**, der fortæller hvor mange processer, der kan tilgå ressourcen på samme tid, samt funktionerne `wait()` og `signal()` til at ændre semafor værdien.

Semaforer har følgende egenskaber:

- **Semaforværdien** initialiseres til at være ikke negativ.
- `wait()` funktionen formindsker **semaforværdien**. Hvis værdien bliver negativ, blokerer `wait()` processen.
- `signal()` funktionen øger værdien. Hvis værdien er ikke positiv, hentes en proces blokeret af `wait()`, ind i **readykøen**.

Semaforer kan være stærke eller svage. Forskellen på de to består i, hvilket princip, processerne bruger til at tildele adgang til ressourcerne.

En stærk **semafor** benytter FIFO[7] princippet, hvor den, der kommer først, får adgang først. En svag **semafor** er ikke udstyret med et princip for afvikling af processer. En af fordelene ved en stærk **semaforstruktur** er at **udsultning** undgåes.

Semaforer i LegOS

Semaforerne i LegOS er bygget over Dijkstra's algoritme [Nie00]. I LegOS er `wait()` og `signal()` funktionerne kaldt henholdsvis `sem_wait()` og `sem_post()`. Herudover tilbyder LegOS yderligere to funktioner; `sem_trywait()`, der fungerer som en wait, der ikke blokerer, samt `sem_getvalue()`, der giver mulighed for at læse **semaforværdien**. LegOS benytter **semaforer** i hukommelsestyringen, processtyringen, samt i LNP (afsnit 2.3.2).

Konklusion

Det at LegOS bruger **semaforer**, gør at RCX'en aldrig på noget tidspunkt vil låse på grund af **gensidig udelukkelse**, **udsultning**, **deadlock** eller **livelock**.

[7] First in, First Out

2.2.5 I/O Kommunikation

Der vil i det følgende afsnit blive beskrevet de tre muligheder for I/O kommunikation: **Programmed I/O**, **Interrupt driven I/O** og **Direct memory access (DMA)**. I/O kommunikation kan være alt fra printeradgang til disk adgang, på en workstationcomputer. På RCX'en er det kommunikationen mellem A/D konverteren (se afsnit 2.1) og I/O-portene.

Programmed I/O fungerer ved, at processoren kører et program, der indeholder instruktioner, som omhandler en I/O enhed. Processoren udfører instruktionen, der er skrevet således, at processoren får fuld kontrol over I/O enheden. I denne type I/O kommunikation er I/O enhedens eneste opgave at udføre kommandoerne fra processoren. Processoren står for at hente data i hukommelsen til output og gemme de indgående data fra inputenhederne. Processoren har tre typer instruktioner, den kan benytte sig af:

Control benyttes til at fortælle enheden, hvad den skal gøre.

Status benyttes til at aflæse enhedens aktuelle tilstand.

Transfer benyttes til at flytte data mellem enheden og processorregistrene.

Programmed I/O læser et word af gangen, og for hver af disse er processoren nødt til at forblive i tjek status tilstand, for at sikre sig, at hele ordet er overført til enhedens register. Denne fremgangsmådes største ulempe er, at processoren er aktiv unødvendigt og bruger lang tid ved at vente på, at enheden er klar til at udveksle data.

Interrupt driven I/O minder meget om programmed I/O, med hensyn til hvordan overførslen af data fungerer. Der er en tydelig forskel på hvordan processorens ressourcer bliver udnyttet. Som forklaret i programmed I/O, bruges der meget CPU-tid på at vente til enheden er klar til overførsel.

Dette problem er løst i interrupt driven I/O ved at ventetiden ligges over på en device controller, der sættes i læse tilstand af processoren. Herefter lytter controlleren til I/O enheden, og processoren fortsætter med andre opgaver. Når I/O-enheden har data klar, sender controlleren et **interrupt** til processoren. I/O-controlleren afventer herefter en forespørgsel fra processoren. Når processoren har brug for det indlæste data, sættes den igangværende proces i tilstanden **waiting**, og de pågældende data fra I/O-enheden overføres til hukommelsen. Herefter fortsættes afviklingen af den proces, der blev sat i **waiting**-tilstand.

Direct memory access (DMA) - Denne form for I/O kommunikation fungerer ved, at **DMA** funktionen køres som seperat hardware enhed på bussen, eller som integreret del af et eksisterende I/O modul. **DMA** er den mest CPU besparende form for I/O kommunikation, da CPU'en kun står for at starte **DMA** modulet. Det fungerer ved, at CPU'en sender følgende informationer til modulet:

- Om det er en læse eller skrive opgave
- Hvilken I/O enhed der skal benyttes
- Hvilket hukommelsesområde der skal benyttes
- Antallet af ord der skal behandles

Herefter klarer modulet selv resten, og CPU'en kan fortsætte med det arbejde, den var i gang med. Modulet begynder nu på at overføre hele hukommelsesblokken, et word af gangen. For at dette kan lade sig gøre, er **DMA** modulet nødsaget til at tage fuld kontrol over bussen. Dette kan godt give en forsinkelse for processoren, hvis den skal bruge bussen i mellemtiden, for den er nødt til at vente på, at **DMA** modulet er færdigt. Når overførslen er færdig, sender **DMA** modulet et interrupt til CPU'en, CPU'en forstyrres med andre ord kun kort under start og i slutningen af overførelsen.

I/O i LegOS

LegOS benytter sig af **interrupt driven I/O** til styring af A/D konverteren, på den måde at A/D konverteren smider et **interrupt** når en konvertering er færdig. Til styring af outputportene bruger LegOS **memory mapped I/O**. Dette foregår på den måde, at der skrives til et sted i hukommelsen for at styre outputporten.

LegOS stiller I/O kommunikation og kontrol til rådighed for motor, sensor, lyd, LCD, samt knapper. Hvordan de enkelte I/O enheder behandles, forklares kort i det følgende afsnit.

Motorerne er kontrolleret af `dm_handler`'en[8]. Dennes job er i dette tilfælde at skifte mellem de forskellige motorer, så de behandles på tur. De mulige motorer er navngivet efter de tre porte A, B og C (herefter angivet som `X`).

Kontrollen foregår ved, at skrive til registeradressen 0xF000 (memory-mapped I/O). Der benyttes to funktioner til dette; `motor_X_dir` som sætter retningen, der kan være:

[8]En handler er i dette tilfælde en særlig del af device driveren

- off, hvor motoren ikke kører, men hjulene kan rotere frit,
- forward, hvor motoren kører fremad
- reverse, hvor motorene kører baglæns
- brake, hvor hjulene er stoppede og ikke kan rotere.

Funktionen `motor_X_speed` sætter motorens fart. Denne betegnes som en værdi imellem `MIN_SPEED` (som er 0) og `MAX_SPEED` (der højst kan være 255).

Sensorer LegOS kan håndtere tre forskellige typer af sensorer; lys-, tryk- og rotationssensorer. Desuden kan det også aflæse batteriniveauet. Dette er med de device drivere, der er inkluderet i LegOS. Med "hjemmelavede" device drivere kan LegOS startset håndtere en vilkårlig ekstern enhed.

Efter A/D konverteren har skrevet de rå sensordata i hukommelsen kan disse aflæses. Til dette benyttes makroerne `SENSOR_1 - SENSOR_3`, en for hver port. Data oversættes herefter af en makro, som er tilpasset de forskellige sensortyper, dette er henholdsvis `LIGHT1-3`, `TOUCH1-3`.

`LIGHT_1 - LIGHT_3` benyttes til lyssensorerne. Disse data oversættes til en værdi mellem 0 og `MAX`, hvor 0 er helt sort og `MAX` er helt hvidt. Tryksensorerne aflæses af `TOUCH_1 - TOUCH_3`. Disse oversætter de rå data til en af værdierne 1 og 0, hvor 1 betyder at sensoren er aktiveret og 0 betyder, at den ikke er aktiveret. Til aflæsning af batteriet benyttes makroen `BATTERY`.

Rotationssensoren er lidt anderledes end de andre sensorer, da denne er udstyret med sin egen handler[9], `ds_rotation`, som bliver kaldt af `ds_handler`. Handleren `ds_rotation_set` bruges til at aflæse den aktuelle værdi, som benyttes som startpunkt for beregning af rotationsvinklen. Herefter beregnes rotationerne af en tilstandsmaskine, hvis funktionalitet vi ikke vil komme nærmere ind på her.

Lyd håndtereres af lydhandleren, der styrer lyden ved enten at afspille en tone eller en pause (det vil sige ingen lyd). Tonerne kan have forskellige længde, whole, half, quarter eller eight, hvor tiden på 1/16th. er 200 millisekunder. Tonerne findes i et array kaldet `pitch2freq`. Afspilningen foregår ved, at funktionen `play_freq` løber arrayet igennem, og bruger systemtimeren til at sende signaler til højtaleren.

LCD styres af LCD-handleren. Der stilles forskellige funktioner tilrådighed for at skrive på displayet:

[9] Behandles seperat af en særskilt del af device driveren

- `cputs` benyttes til at udskrive strings
- `cputw` udskriver hexadecimal ord
- `cputc_native` udskriver enkelte hexnumre
- `cputc` benyttes til at vise enkelte ASCII værdier

Herudover findes funktion `cls`, der benyttes til at resette displayet. Opdateringen af displayet kan foretages manuelt ved at benytte funktion `lcd_refresh`, som udskriver de 9 bytes der er gemt i arrayet `display_memory`. Endnu et array benyttes, `lcd_shadow`, som indholder de sidste bytes, der er skrevet på displayet og bruges til at tjekke om en opdatering er nødvendig. Handleren opdaterer displayet automatisk ved at kalde funktionen `lcd_refresh_next_byte`, hvert 6. millisekund, og da LCD dataene er 9 bytes lange, sker der en komplet opdatering hvert 54. millisekund, det svarer til en 18 - 19 gange i sekundet.

Knapperne håndteres af handleren `dkey_handler`. Handleren er debounced, hvilket betyder, at knappernes tilstande kun tjekkes når tiden er udløbet. For at undgå at mere end et tryk på knappen bliver aflæst af gangen, sættes tiden til 100 ms. Når handler funktionen kaldes undersøges tiden. Hvis denne er 0 aflæses knappernes tilstande, ellers bliver de ikke aflæst.

Der er 4 knapper på boksen, der er koblet til I/O-portene på følgende måde: **on/off** og **run** knapperne er koblet på I/O port 4 på henholdesvis bit 1 og 2. **View** og **prgm** på port 7 bits 6 og 7. Disse 4 bits danner tilsammen en bitmask. Informationer om bitmasken gemmes i variablerne `dkey_multi`, der indikerer, hvilken bitmask, der er brugt sidst. Variablen `dkey` fortæller, hvilken knap, der er trykket på.

Til at aflæse knapperne benyttes funktionen `getchar`. De data, som `getchar` indhenter, oversættes af følgende makroer, `KEY_ONOFF`, `KEY_RUN`, `KEY_VIEW`, `KEY_PRGM` samt `KEY_ANY`.

2.2.6 Andre RCX operativsystemer

Der vil i det følgende blive kigget nærmere på to andre RCX operativsystemer; nemlig NossOS og ChaOS. Begge er udviklet på AAU, henholdsvis i 2000 og 2001. ChaOS og NossOS er designet på to forskellige måder, og vi vil komme ind på nogle af de designvalg der er truffet i begge OS'er. Generel operativsystem teori findes i afsnit 2.2. Følgende kilder er brugt i dette afsnit [Ef01] og [gC00].

ChaOS

ChaOS systemet er bygget over en monolithic kerne (se evt. 2.2.1, og benytter en cyclic single linked list til at gemme information om hver enkelt proces. Disse lister danner et hjul og der findes syv hjul, et pr. prioriterings mulighed. Et hjul kan i teorien indeholde 1785 processer, men der er en hukommelses begrænsning. Hver proces har et **Proces ID** (**PID**), der er en pointer til processens placering i hukommelsen[Ef01]. Hjulene benytter 3 variabler til at holde styr på tingene.

- `Total_processes`, som holder styr på, hvor mange processer, der er i hvert hjul.
- `previus_process`, som indeholder information om, hvilken proces, der lige er blevet kørt.
- `current_process`, som indeholder information om hvilken proces, der skal køres næste gang.

Process Manageren stiller nogle funktioner til rådighed for brugerprogrammer. Disse funktioner er `new_process()` og `kill()`, der som navnet antyder kan starte og dræbe processer.

Semaforerne i ChaOS er binære og interfacet benytter tre routiner `Wait`, `Signal` og `try_wait`

- `Wait` benyttes til at undersøge om det er muligt at komme ind i den region som semaforen beskytter.
- `Signal` benyttes til at fortælle den næste proces i køen, at der er frit.
- `Try_wait` er en forespørgsel, som benyttes til at undersøge om regionen kan benyttes med det samme.

Hukommelsesstyringen i ChaOS er bygget over worst fit algoritmen, som er beskrevet i afsnit 2.2.2

NossOS

Funktionen `task_new()` står for at oprette nye processer, funktionen benytter kun et enkelt argument, det valgte navn. For at indsætte en ny process i processlisten, benyttes `element_add()`. NossOS benytter kun een procesliste. Funktionen `element_add()` benytter 3 argumenter, hvilken procesliste, der skal be-

nyttes, navnet på processen og proces id. Processer kan være i en af følgende fem tilstande.

- `State_init` processen er oprettet.
- `State_Ready` procesen kører.
- `State_Blocked` processen er blokeret.
- `State_Waiting` processen venter på en hændelse.
- `State_Killme` processen er klar til at blive nedlagt.

Scheduleringen foregår uden brug af prioriteringsmuligheder, processerne afvikles ved, at task switcheren tjekker om processen benytter nogle IPC funktioner. Er dette tilfældet, får processen lov at afslutte, ellers findes den næste proces i listen, der er i INIT eller READY tilstanden, og denne afvikles.

2.3 Netværksteori

I det følgende afsnit vil vi redegøre for generel netværkteori, med udspring i en analyse af **LegOS Network Protokol** (**LNP**), som er den netværksprotokol, der er implementeret i LegOS (version 0.2.6). Analysen af LNP og gennemgangen af generel netværksteori, skal være en støtte i designet af vores egen netværksprotokol.

Da LNP er en lagdelt protokol, vil vi forsøge at knytte den op på OSI-modellen, som er en referencemodel til design af lagdelte netværksprotokoller. Som primær kilde til dette afsnit er brugt [Tan96]

2.3.1 Et computernetværk

Et computernetværk består af enheder koblet sammen, således at de kan kommunikere indbyrdes og på tværs af andre netværk.

De fleste netværk er **Multiple access networks**, som betyder, at alle kan kommunikere med alle via et fælles medie, for eksempel en **hub** eller en **switch**.

Ved at koble flere netværk sammen ved hjælp af en **router**, opstår et **internetwork**.

Adressering

Det er vigtigt at kunne adressere de forskellige **hosts** i et netværk. Til dette bruges en streng af bytes, som giver hver **hosts** en unik adresse. På internettet bruges en service kaldet **Domain Name Service** (**DNS**) til at oversætte de let genkendelige adresser, som for eksempel www.lego.dk, til en computers **Internet Protocol** (**IP**) adresse, for eksempel 130.225.194.27.

Adressering i LNP foregår ved på forhånd at definere, hvad den enkelte RCX's adresse er. Dette gøres ved at sætte variablerne `LNP_HOSTADDR` og `LNP_HOSTMASK`. `LNP_HOSTMASK` bestemmer, hvor meget af adresseheaderen, der skal bruges til selve hostnavnet. De resterende bliver brugt til at fortælle, hvor mange porte de enkelte hosts har. `LNP_HOSTADDR` er hostnavnet, som skal skrives ind i kildekoden for LegOS for hver enkelt RCX.

Generelt bruges en **subnet mask** til at nå maskiner i et internetværk. Hvis en datapakke sendes til en given IP-adresse, så vil subnettets router lave en **AND operation** mellem IP-adressen og subnet masken, for at få det netværksnummer, som den pågældende IP-adresse ligger på. Dette er smart, fordi en router ikke behøver at kende andet, end adresserne på de andre subnets og selvfølgelig maskinerne på

sit eget subnet. Dette vil være usynligt udadtil, og betyder blandt andet også, at hvis et netværk vokser, er der ingen globale databaser, som skal opdateres.

Performance

Et andet vigtigt aspekt omkring netværk er netværkets performance. Performance eller ydelse kan måles på flere måder. Først og fremmest er der båndbredde. Båndbredden er et udtryk for, hvor meget data, der kan sendes/modtages per tid. Båndbredden måles ofte i bit per sekund.

Netværkets ydelse kan også udtrykkes ved hjælp af den tid, det tager for en pakke at nå fra en ende af netværket til den anden og tilbage igen. Dette kaldes for **Round Trip Time** (**RTT**) og måles i millisekunder.

Design af et netværk

Under design af et netværk og de tilhørende protokoller, er der adskillige ting, der bør tages i betragtning, for at opnå en holdbar arkitektur. Dette kan opsummeres til en række krav, som netværket bør overholde.

En netværksarkitektur skal:

- være generel
- være effektiv i forhold til prisen
- være fair
- være robust
- have en god ydelse
- have mulighed for at forbinde mange computere
- tage højde for ændringer i teknologien
- være åben overfor nyt software

Første del af at designe en netværksarkitektur er at vurdere, hvilke af disse krav, netværket skal leve op til. For eksempel er det vigtigt, at et netværk til styring af pengetransaktioner er robust, hvorimod det er mindre vigtigt, at det er generelt, da det kun skal bruges til dette specifikke formål.

LNP er baseret på en lagdelt arkitektur. Den lagdelte struktur har til formål at indkapsle forskellige dele af arkitekturen. Hvert enkelt lag skal have et interface til det underliggende og overliggende lag. Dette interface skal definere en række primitive operationer og services, som det underliggende lag stiller til rådighed for det overliggende.

En fordel ved denne arkitektur er, at veldefinerede interfaces gør det lettere at udskifte et enkelt lag. Der er en lang række kriterier, som gør sig gældende, når lagene designes. Det gælder for eksempel adressering, fejlkontrol, routing og så videre.

2.3.2 OSI-modellen

I det følgende redegøres for OSI modellen (Open Systems Interconnection), som er en referencemodel til design af netværksprotokoller. Senere vil vi anvende begreber og udtryk fra denne model til at beskrive og analysere LNP. Når der bruges andre tekniske ord fra den relevante netværksteori vil disse forklares. De anvendte kilder er [LV02],[Nie00],[Vil00] og [Tan96],[Chr02].

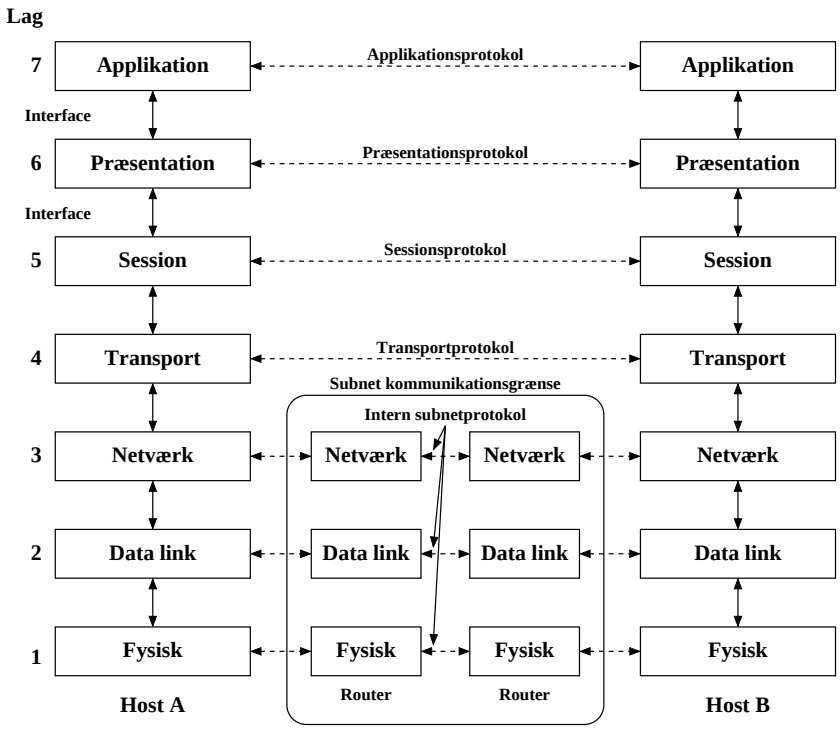
En vigtig lagdelt referencemodel til kommunikation mellem systemer er OSI modellen. OSI modellen er udviklet i 1983, og er delt op i syv lag. De syv lag er fremkommet udfra fem grundlæggende principper, der siger:

- at der skal være et lag for hvert niveau af abstraktion
- at hvert lag skal udføre en veldefineret funktion
- at lagene skal designes med henblik på internationale standarder
- at veldefinerede interfaces skal minimere dataudveksling mellem lagene
- at antallet af lag skal vælges, således at forskellige funktioner ikke blandes sammen i det samme lag.

Figur 2.10 på side 37 fra viser OSI reference modellen med de syv lag, som kort er beskrevet nedenfor. Ligeledes vil vi knytte LNP's lag op på OSI modellens lag.

Det fysiske lag

Dette lag har til opgave at sende og modtage rå data bits. Designvalg omhandler problemstillinger omkring mekaniske og elektriske aspekter. Dette kan for eksempel være, hvor mange volt, der skal representere en 1 bit og en 0 bit, eller hvor lang



Figur 2.10: OSI modellen inspiration [Tan96]

tid en bit varer. I LNP udgøres det fysiske lag af IR porten, A/D konverteren, **SCI** (Serial Communication Interface, se afsnit 2.1, side 10) og registre.

Ydermere har LNP et såkaldt logisk lag. Det passer ikke ind i referencemodellen til OSI modellen. Det hører til et sted mellem det fysiske lag og data link laget. Det logiske lags funktion er at sende bits ind i **SCI** og ligeledes trække bits ud og stille dem til rådighed for det ovenliggende lag.

I et "almindeligt PC-netværk" ville der sædvanligvis være et netværkskort til at varetage den slags funktioner. Men RCX'en har sparsomme ressourcer, og derfor bliver den nødt til at implementere dette i software. Derfor har den det logiske lag som et ekstra lag lige over det fysiske lag.

Data link laget

Data link laget skal stille et veldefineret interface til rådighed for netværkslaget. Laget skal endvidere stå for inddeling af bits i rammer, fejlhåndtering (herunder styre timere, sekvensnumre og beregne checksum) og regulering af hastigheden

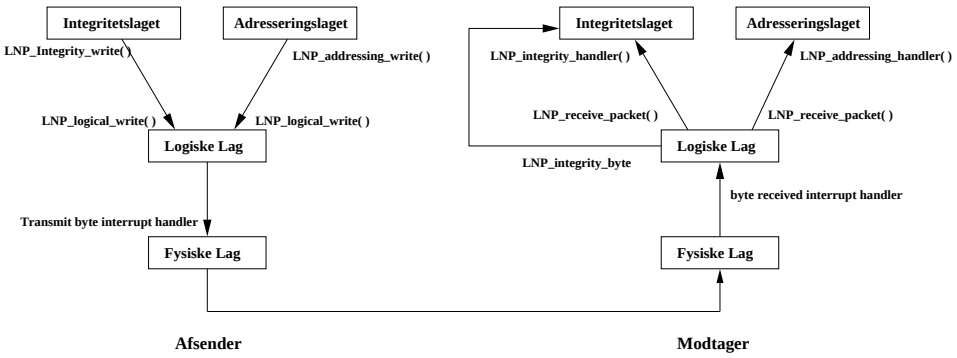
(flow control). Tre typiske tjenester, som data link laget kan stille til rådighed for det overliggende netværkslag er beskrevet nedenfor.

- **Ubekræftet forbindelsesløs tjeneste** Bruges til data, hvor tidsfaktoren er vigtigere end at alt data når frem. For eksempel tale- og videostreaming.

- **Bekræftet forbindelsesløs tjeneste** Bruges til upålidelige medier, for eksempel trådløse, hvor tabt data bliver gentransmitteret.

- **Bekræftet forbindelsesorienteret tjeneste** Denne tjeneste opretter en forbindelse, som afbrydes efter brug, til transmission af data. Den garanterer, at data når frem. Denne type tjeneste er den mest brugte til kommunikation på internettet.

Data link laget i LNP svarer til de følgende to lag: integritets- og adresseringslaget.



Figur 2.11: LNP protokolstakken

Ved at bruge integritetslaget kan der sendes pakker, som ikke indeholder nogen form for adresseringsinformation og derved bliver broadcastet til alle modtagere.

Den anden form for pakke, der kan sendes, er adresseringspakker. Disse bliver sendt fra adresseringslaget, og her er der tilføjet adresseringsinformation, såsom afsender og modtager, og pakkerne kan blive leveret til en specifik port/service på en specifik enhed. Adresseringspakker er LNP's måde at tilbyde en ubekræftet forbindelsesløs tjeneste på.

Integritets- og adresseringslaget sender begge to deres pakker ned til det logiske lag. Derfor ligger de på samme niveau som det ses på figur 2.11.

Netværkslaget

Dette lag står for at route pakker fra sender til modtager. Endvidere skal laget forsøge at kontrollere strømmen af data (congestion control), for at undgå flaskehalse. Netværkslaget bruges også af internetudbyderen til at samle information om kundens brug af tjenester. Denne information kunne bruges til afregning.

Når pakker skal routes over forskellige netværk for at nå fra sender til modtager, opstår der en række problemer. For eksempel kan to netværk bruge forskellige måder at adressere på. Disse aspekter skal være usynlige for transportlaget, og det er derfor vigtigt, at netværkslaget tilbyder et veldefineret interface.

I et **broadcast** netværk, som er et netværk, hvor alt data sendes ud til alle, vil netværkslaget ofte være tyndt/ikke eksisterende, da pakkerne ikke bliver routet.

Transportlaget

Et godt designet transportlag tillader programmer at kommunikere sammen, uden at skulle tænke på den underliggende hardware. Det vil sige, at transportlaget er forbindelsesled mellem programmer, og netværkslaget er forbindelsesled mellem maskiner. Transportlaget modtager data fra sessionslaget, deler det op i mindre dele, og giver dem videre til netværkslaget.

Transportlaget er et sandt “end-to-end” lag. Med andre ord, et program på afsendermaskinen snakker med et tilsvarende program på modtagermaskinen.

Endvidere skal transportlaget tilbyde en form for flow kontrol, for at undgå at en hurtig afsender ikke oversvømmer en langsom modtager med data.

Sessionslaget

Sessionslaget tilbyder udvidede tjenester i forhold til transportlaget. Hvis for eksempel en filoverførsel bliver afbrudt, kan sessionslaget have implementeret metoder til at genoptage forbindelsen.

Præsentationslaget

I modsætning til de lavere lag, der koncentrerer sig om at flytte bits, rammer og pakker, flytter præsentationslaget rundt på karakterer, strenge, tal og forskellige

data typer. Præsentationslaget gør det muligt for computere, der har forskellige måder at representere data typer på, at udveksle data.

Applikationslaget

Øverst i OSI modellens protokolstak ligger applikationslaget. Dette lag indeholder en række kendte og ofte anvendte protokoller. Dette er protokoller som FTP, telnet, SMTP og DNS. Som et eksempel bruges FTP protokollen til at overføre filer mellem maskiner, og den skal være designet til at tage højde for de problemer, som opstår ved overførsel mellem forskellige systemer.

Et Unix-baseret system har for eksempel en anden måde at lave linieendelser i filer på end et Microsoft Windows baseret system, der skelner mellem tekstfiler og binære filer [10].

2.3.3 Sliding Window Protokol

I datalink laget er en af de mest udbredte protokoller den såkaldte **sliding window** protokol. Den giver mulighed for både at sende og modtage på samme tid over den samme kanal, altså er det en **fuld duplex** protokol.

Når der sendes en pakke, skal modtageren bekræfte, at den har fået pakken. Dette kan gøres ved at sende en seperat **acknowledge** (**ACK**) pakke. En smartere måde at sende **ACK pakken** på, er ved at sende den sammen med eventuelle udgående datapakker fra modtageren. Denne teknik kaldes **piggybacking** og den sørger for bedre udnyttelse af båndbredden. Hvis modtageren ikke har noget data at sende, bliver den nødt til at sende en seperat **ACK pakke**.

Hver gang der sendes en pakke, sættes en timer i gang. Hvis timeren løber ud, betragtes pakken som tabt, og den pågældende pakke vil blive forsøgt retransmitteret. Dette kaldes et **timeout** for en pakke. Det er vigtigt, at timeoutet ikke indtræffer for tidligt, da modtageren muligvis ikke når at sende en **ACK** pakke tilbage, og forbindelsen vil derved blive oversvømmet med retransmissioner. Hvis **timeout**'et indtræffer for sent, bliver båndbredden ikke godt udnyttet. Generelt vil det være optimalt, hvis timeoutet er lidt længere end **RTT** tiden, da modtageren derved har lidt tid til at sende sit **ACK**.

Hver pakke, der bliver sendt, skal også have et sekvensnummer, så modtageren ved, at den får de rigtige pakker. Hvis den får en pakke med et forkert sekvensnummer vil den ignorere pakken, som efterfølgende vil blive retransmitteret.

[10] Diverse unices bruger newline, hvor Windows bruger både carriage return og newline

En af nøglefunktionerne ved en **sliding window** protokol, er at senderen altid ved, hvor mange pakker, den må sende ud, da den altid opretholder en liste over, hvor mange pakker, den har sendt, og hvilke af disse, der er blevet bekræftet. Denne liste kaldes også for senderens vindue. Ligeledes opretholder modtageren en liste over, hvilke pakker den må acceptere, altså modtagerens vindue.

I nogle af protokollerne er størrelsen på disse vinduer fast, mens den kan variere i andre. Når senderen modtager en pakke fra det ovenliggende lag får den det højeste sekvensnummer, og størrelsen af senderens vindue bliver forøget med en. Når senderen får en **ACK** tilbage, formindskes vinduet med en. Modtagerens vindue bliver 1 større, når den modtager en pakke og 1 mindre når den sender en **ACK** tilbage.

En anden smart teknik, som også bruges, i mere avancerede protokoller, er **pipelining**. Hvis to maskiner skal kommunikere sammen over meget lang afstand, bliver den såkaldte **RTT** en meget vigtig faktor, som vi i det forrige eksempel så helt bort fra. Hvis der skulle ventes på bekræftelse på hver eneste pakke, som for eksempel skulle ud i rummet, ville **RTT** være forholdsmæssigt stort. Altså ville senderen stå og vente i det meste af tiden, før den kunne sende den næste pakke. Dette ville give meget dårlig udnyttelse af båndbredden. Derfor øges størrelsen på vinduet så senderen får lov til at sende adskillige pakker, før den får bekræftelse på den første pakke. Størrelsen på vinduet bør være tiden en pakkes **RTT** tager, for at det fungerer optimalt. Heraf kommer ordet **pipelining**, da det rent begrebsmæssigt kan betragtes som et rør, som pakkerne propagerer igennem.

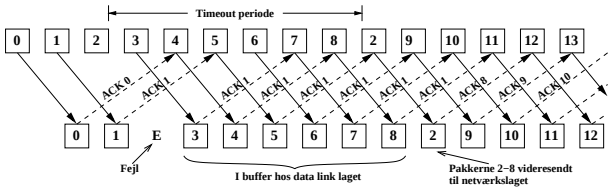
Selective repeat sliding window

I det følgende vil vi give et eksempel på en **sliding window** protokol, nemlig **selective repeat**.

Både senderens og modtagerens vindue kan variere i denne udgave af sliding window protokollen. Dermed kan senderen starte med at sende x antal pakker, uden at få en **ACK** tilbage. Disse ubekræftede pakker skal buffereres, og må først slettes, når senderen har fået en bekræftelse tilbage for hver enkelt.

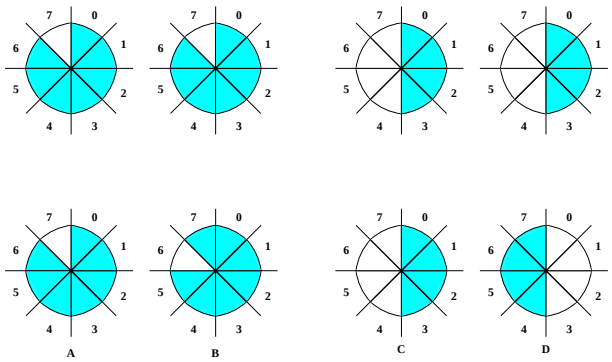
Modtageren skal også bufferere pakker, men det skal kun ske, når der er sket en fejl - det vil sige, når den får pakke med et sekvensnummer, der ikke efterfølger det sekvensnummer, den foregående pakke havde. Hvis afsenderen sender mange pakker, og der for eksempel sker en fejl midtvejs, skal modtageren bufferere de efterfølgende pakker, indtil dens vindue er fyldt op, da den stadig kun må acceptere pakker, som er indenfor vinduet. Den pakke som modtageren ikke fik, vil efterfølgende udløse et **timeout**, og blive retransmitteret igen. Når modtageren får den manglende pakke, vil den derfor have mange korrekte pakker, som den hurtigt kan

give videre til netværkslaget (se figur 2.12). Senderens og modtagerens vinduer må



Figur 2.12: Håndtering af pakketab i **Selective repeat**.

aldrig være større end halvdelen af antallet af sekvensnumre. Derved vil senderens "nye" vindue aldrig overlappe det gamle, så den ikke sender forkerte pakker, hvis der er opstået en fejl. Som det ses på figur 2.13 (side 42) vil det gå galt i situation a og b, når senderens vindue har størrelsen 7 ud af 8 sekvensnumre. For hvis modtageren accepterer de første 7 pakker, og sender **ACK** pakker tilbage, men hvis disse **ACK** går tabt, vil senderen gentransmittere de første 7 pakker, som modtageren vil tro er nye pakker (da de ligger inden for dens nye vindue). Protokollen vil dermed fejle med de vinduestørrelser. Situationen i c og d vil derimod blive håndteret korrekt, da senderens vindue kun er halvdelen af sekvensnumrene. Hvis modtagerens **ACK** går tabt, vil senderen gentransmittere de første 4 pakker, som ikke vil falde indenfor modtagerens vindue.

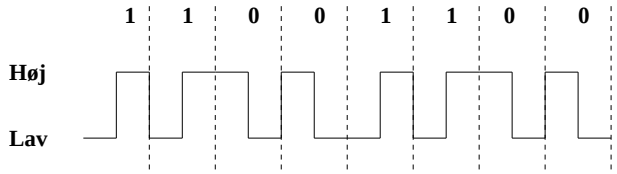


Figur 2.13: Problemstillingen ved størrelsen på vinduerne i **Selective repeat**.

Alternating Bit Denne protokol er forgængeren for **sliding window**. Den går i sin simpelthed ud på, at det sekvensnummer som pakken får, kun veksler mellem 0 og 1, altså sekvensnummeret er en enkelt bit.

2.3.4 Manchester Encoding

Når to maskiner skal sende bits til hinanden, er det selvfølgelig vigtigt, at de kan skelne mellem en 0 og en 1 bit. Da vi i vores egen netværksprotokol har tænkt os at sende bits ved at sende x volt ud på kablet, ville det være en oplagt mulighed at bruge **Manchester Encoding** (**ME**). I stedet for at sende for eksempel 5 volt (høj) for 1 bit og 0 volt (lav) for 0 bit, fungerer **ME** ved at repræsentere en 0 bit som høj efterfulgt af lav. Tilsvarende repræsenteres en 1 bit som lav efterfulgt af høj (se figur 2.14). Der er altså en overgang mellem en bit, og derved er modtageren ikke i tvivl om, at den modtager en 0 bit, eller senderen bare ikke transmitterer noget (derved ville den jo også modtage 0 volt). En stor ulempe ved **ME** er selvfølgelig, at den halverer båndbredden.



Figur 2.14: Repræsentation af bits ved hjælp af **Manchester Encoding**.

2.3.5 Fejlhåndtering

For at være sikker på, at de data, der bliver sendt over netværket er korrekte, bruges en eller anden form for fejlhåndtering. Denne deles som regel op i to kategorier, nemlig fejldetektering og fejlkorrektion.

Fejlkorrektion

Når der sker fejl under datatransmissionen på et netværk, kan disse fejl korrigeres ved hjælp af den såkaldte **Hamming distance** (der er beskrevet i [Tan96]). Et **codeword** består af en mængde bits og nogle checkbits. Den førnævnte distance beregnes ved at se på, hvor mange bits der er forskellige ved to **codewords**, som kan beregnes ved hjælp af **EXCLUSIVE OR** (**XOR**) operationen. Hvis to **codewords** afviger fra hinanden med en **Hamming distance** d, vil det derfor kræve d bitfejl at lave den ene om til den anden.

Det er muligt at konstruere en komplet liste af alle gyldige **codewords**, ud fra algoritmen[11] til at beregne check bits med. Ydermere skal den mindste d mellem

[11] For ikke at skulle forklare alt for mange perifære tekniske detaljer, har vi valgt ikke at dvæle ved

to **codewords** findes ud fra denne liste, og dette er så **Hamming distancen** af hele koden.

For at detektere h fejl, skal distancen bare være større end antallet af fejl, altså $d = h + 1$. For at rette h fejl skal $d = 2h + 1$. Derved bliver listen af **codewords** så forskellige fra hinanden, at selv med h fejl vil det oprindelige **codeword** stadig være tættere på end nogen af de andre. **Hamming koder** kan kun rette enkelte bitfejl. Det er dog muligt at rette **burst errors**, ved at arrangere **codewords** i en matrice, men det vil vi ikke komme nærmere ind på.

Fejldetektering

I stedet for fejlkorrektion bruges ofte fejldetektering og retransmission. Dette gøres ved både at detektere kollissioner, samt ved at beregne en checksum for det data, der udveksles.

Til at detektere deciderede kollisioner på netværket, bruges **Carrier Sense Multiple Access with Collision Detection** (**CSMA/CD**). Hvis to eller flere enheder på netværket forsøger at sende data samtidigt, opstår en kollision. En enhed kan detektere en sådan kollision ved at sammenligne det data den sendte, med den, der modtages. Er disse forskellige, afbrydes afsendelsen af data øjeblikkeligt og enheden venter et tilfældigt tidsrum. Når denne tid er gået, prøver den at sende sit data igen, såfremt ingen andre enheder i mellemtiden er begyndt at sende data.

Kollisionsdetektering i LNP foregår på følgende måde:

```
1   lnp_timeout_reset ();
2   if (tx_state <TX_ACTIVE) {
3     // foreign bytes
4     //
5     allow_tx=sys_time+LNP_BYTE_SAFE;
6     lnp_integrity_byte (S_RDR);
7   } else {
8     // echos of own bytes -> collision detection
9     //
10    if (S_RDR!=*tx_verify) {
11      txend_handler();
12      tx_state =TX_COLL;
13    } else if ( tx_end <= ++tx_verify ) {
14      // let transmission end handler handle things
15      //
16      tx_state =TX_IDLE;
17    }
18  }
```

Ovenstående kontrollerer, om den pakke den får ind er magen til den, den sen-

denne algoritme

der. Opdager den en kollision, kalder den `txend_handler()`, som straks afslutter transmissionen. For at undgå, at to maskiner hele tiden blokerer hinanden, altså et **livelock**, bruger den **CSMA/CD algoritmen**[Chr02]. Hver gang funktionen har modtaget en byte, undlader den at sende noget i et tidsrum, der er defineret i `LNP_BYTE_SAFE`. Ligeledes undlader LNP at sende noget (defineret i `LNP_BYTE_SAFE`), lige efter den har sendt en pakke. Derved giver den andre maskiner mulighed for at sende data.

Hvis en kollision skulle ske, vil den straks stoppe transmissionen og undlade at sende data i den periode som er givet ved allow_tx $= 12 *$ LNP_BYTE_TIME $+$ randomvalue. Det er vigtigt, at der bliver lagt en tilfældig værdi til. Hvis dette ikke bliver gjort, vil der være mulighed for, at de to maskiner, som kolliderer, vil vente den samme periode, hver gang og derved være i **livelock**.

En checksum [Tan96] beregnes udfra det data, der skal sendes, og denne checksum bliver sendt med pakken til modtageren, som dernæst kontrollerer, om checksummen passer.

Polynomiske koder udregnes udfra bit-strenge, hvor for eksempel 110001 har 6 bits og vil blive repræsenteret som et femtegrads polynomium således:

$$x^5 + x^4 + x^0$$

Det beregnede polynomium skal tilføjes til slutningen af pakken, således at polynomiet kan divideres med $G(x)$, hvor $G(x)$ er et **generator polynomial**. For at dividere to polynomier med hinanden, bruges der **XOR**. Det vil sige, at når modtageren får pakken, dividerer den checksummen med **generator polynomial**, og hvis der findes nogen rest fra divisionen, har der været en transmissionsfejl.

Algoritmen til at beregne checksummen er som følger:

- Lad r være den højeste grad af $G(x)$. Tilføj r nulbits til slutningen af pakken, så den indeholder $m + r$ bits og svarer til polynomiet $x^r M(x)$.
- Divider bitstrengen, der svarer til $G(x)$ med den tilsvarende bitstreng, som svarer til $x^r M(x)$.
- Resten fra ovenstående division er checksummen, som skal tilføjes til slutningen af det egentlige data.

Fejlhåndtering i LNP

I LegOS bliver checksummen beregnet af følgende funktion i LNP:

```
1  unsigned short lnp_checksum(
2    const unsigned char *data,
3    unsigned length
4  )
5  {
6    unsigned char a = 0xff;
7    unsigned char b = 0xff;
8
9    while (length > 0) {
10     a = a + *data;
11     b = b + a;
12     data++;
13     length--;
14   }
15   return a + (b << 8);
16 }
```

Checksummen beregnes udfra ASCII-værdien af hvert tegn i **data**-arrayet, der har offset i `data`. Disse værdier bliver lagt sammen i variablen `a`. `a` er af char-typen, som rummer værdierne $0..255$. Overstiger summen af `a` 255, startes forfra med den overskydende værdi. Eksempelvis vil $254 + 25$ være lig med 23. `a`'s startværdi er 255.

Efterfølgende regnes en `b`-værdi ud. Denne værdi er lig med værdien af `a`, plus den tidligere værdi af `b`. Endeligt lægges værdien af `a` til værdien af `b`, der bliver bitshiftet otte gange til venstre.

Vi har ikke været i stand til at finde nogen officiel algoritme, som denne checksum er beregnet ud fra. Vi antager, at Marcus L. Noga selv har fundet på den.

Kort sagt ligges char-værdierne sammen adskillige gange og det samlede resultat bruges som checksum. Dermed er det antal forskellige værdier som checksummen kan give 2^{16}. Da checksummen, der bliver sendt med, kun kan have størrelsen **byte**, bruges kun de 8 højestbetydende bits i checksummen. Dette giver 256 forskellige værdier, og der er derfor en risiko for, at en checksum for en korrekt pakke, og en checksum for en fejlbehæftet pakke er ens.

Afsluttende overvejelser

Afslutningsvis skal det siges, at der meget ofte vælges fejldetektering og retransmission frem for fejlkorrektion. Eksempelvis kræver det 10000 ekstra checkbits til fejlkorrektion ved brug af **Hamming koder** til at sende 1 Mbit data (såfremt pakkestørrelsen er på 1000 bits), mens det kun vil kræve 1000 checkbits at detektere en fejl. Ved pakkeafsendelse i LNP er det ikke er sikkert, at pakker når frem - men hvis de gør, er det næsten garanteret, at de er fejlfri. Når pakken ikke når frem/er fejlramt, får afsenderen det ikke at vide.

LNP gør dermed hverken brug af retransmission eller fejlkorrektion. Der er derfor altid en risiko for, at det sendte data går tabt. Da det er infrarød kommunikation har omgivelsernes påvirkning stor indflydelse på denne risiko. Såfremt omgivelserne er frie for støj (sollys og andre former for infrarød stråling), vil langt de fleste transmissioner ske uden fejl. Den største faktor vil sædvanligvis være for eksempel kraftigt lys eller blokerende objekter mellem sender og modtager.

2.3.6 Protokolstakken i LNP

For at opnå forståelse for, hvordan en netværksprotokol til RCX'en virker, har vi valgt at bruge et realistisk eksempel, hvor vi vil se på den vej, en datapakke bevæger sig gennem LNP protokolstakken. På figur 2.11 på side 38 ses en oversigt over protokolstakken - og de vigtigste funktioner, der dirigerer data fra afsenderen til modtageren.

Opstart af LNP

LNP startes ved at `program_init()` i filen *program.c* bliver kaldt fra *kmain.c* hvis `CONF_PROGRAM` i *config.h* er defineret, og det er den som standard.

```
1  void program_init () {
2     packet_len =0;
3     sem_init (&packet_sem,0,0);
4     execi (&packet_consumer,0,0,PRIO_HIGHEST,DEFAULT_STACK_SIZE);
5     execi (&key_handler,0,0, PRIO_HIGHEST,DEFAULT_STACK_SIZE);
6     lnp_addressing_set_handler  (0,&packet_producer);
7  }
```

Her bliver `packet_consumer()`'en startet som en tråd, sammen med `key_handler()`'en. De to tråde styrer den dynamiske programindlæsning. Begge får den højeste prioritet, og vil således blive udført hyppigt. `packet_consumer()`'en håndterer indgående trafik på IR porten, og `key_handleren()` håndterer tastetryk.

LNP bruger en delt buffer, som er beskyttet af en semafor, kaldet `packet_sem`. Det bevirker at `packet_consumer()` står og venter på, at `packet_producer()`'en har kopieret inddataen til bufferen[12]. Packet produceren bliver startet af et kald fra adresseringslaget[13]. Den læser dernæst den kommando, som pakkens header indeholder. De mulige kommandoer er **Delete**, **Create**, **Data** og **Run**.

[12] Læs mere om dette i afsnit 2.2.4

[13] Dette lag er beskrevet i afsnit 2.3.2 på side 37

De følgende header informationer bruges kun, når der skal uploades brugerprogrammer til RCX'en. Hvis pakkens første headerfelt indeholder

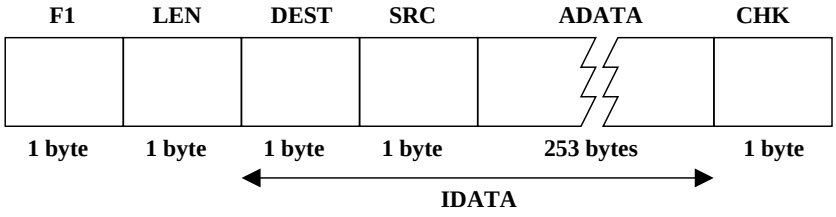
- **Delete** laver `packet_consumer()`'en plads til programmet i **user program array** igennem user interfacet[Nie00].
- **Create** får `packet_consumer()`'en til at allokere hukommelse for programmet, hvor den nødvendige hukommelsesplads står beskrevet i pakkens header.
- **Data** får `packet_consumer()`'en til at kopiere hele pakken, eksklusiv headeren, ind i den allokerede hukommelsesblok.
- **Run** begynder at udføre programmet, som er lagt ind i hukommelsesblokken.

Hver gang en pakke er korrekt modtaget og kommandoen er udført, sender `packet_consumer()`'en en bekræftelse tilbage til afsenderen, og dette sker kun, når der er tale om upload af programmer.

lnp_addressing_write

Når det drejer sig om almindelig netværkstrafik (det vil sige, når det ikke gælder upload af programmer), starter en pakke sin færd gennem protokolstakken fra adresseringslaget i denne funktion. Pakken bliver opbygget efter nedenstående regler, og vil se ud som på figur 2.15. Når der sendes data med `lnp_addressing_write()`, er det kun modtageren i `dest`-feltet, der læser pakkens data.

En adresseringspakke er opbygget på følgende måde (se figur 2.15).



Figur 2.15: LNP adresseringspakke

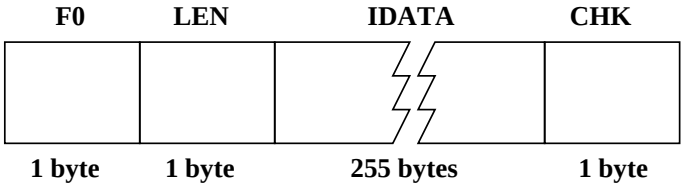
- F1 indeholder en byte, som angiver, hvorvidt det er en adresseringspakke eller en integritetspakke. I dette tilfælde en adresseringspakke.
- LEN angiver længden af det egentlige data som pakken indeholder.
- DEST angiver destinationsmaskinen.

- SRC angiver, hvor pakken blev sendt fra
- ADATA bliver brugt til det egentlige data. Dette data kan maksimalt optage 253 bytes, da der til sidst i pakken skal være plads til checksummen.
- CHK indeholder checksummen, som sørger for, at modtageren kan verificere pakkens korrekthed.

Dette giver en total pakke længde på 258 bytes. Når `lnp_addressing_write()` har bygget pakken op, bliver den sendt videre til `lnp_logical_write()`.

lnp_integrity_write()

En integritetspakke er en **broadcast**-pakke. Det vil sige en pakke, som alle på netværket modtager og læser indholdet af. Den er bygget op på følgende måde (se figur 2.16).



Figur 2.16: LNP integritetspakke

- F0 indeholder en byte som angiver, hvorvidt det er en adresseringspakke eller en integritetspakke. I dette tilfælde en integritetspakke.
- LEN angiver længden af det egentlige data som pakken indeholder.
- IDATA plads og frem bliver brugt til det egentlige data. Dette data kan dog maksimalt fylde 255 bytes, da der til sidst i pakken skal være plads til checksummen.
- CHK indeholder checksummen, som sørger for, at modtageren kan verificere pakkens korrekthed.

Dette giver en total pakke længde på 258 bytes.

`lnp_integrity_write()` opbygger sin pakke efter ovenstående metode, og sender derefter pakken videre til `lnp_logical_write()`.

lnp_logical_write()

Funktionen `lnp_logical_write()` (fra *lnp-logical.c*) bryder de pakker den får fra de øvre lag, op i **frames**/bytes. Ydermere er det den funktion som kommunikerer direkte med RCX'ens **SCI** (se afsnit 2.1, side 10) til at sende en byte**frame** over IR-porten.

```
1  int lnp_logical_write (const void* buf, size_t len) {
2    unsigned char tmp;
3
4  //# ifdef CONF_TM
5  //   sem_wait(&tx_sem);
6  //# endif
7
8  #ifdef CONF_AUTOSHUTOFF
9    shutoff_restart ();
10 #endif
11
12   wait_event(write_allow,0);
13
14   lnp_timeout_reset ();
15
16   tx_verify =tx_ptr=buf;                    // what to transmit
17   tx_end=buf+len;
18
19   tx_state =TX_ACTIVE;
20   S_SR&=~(SSR_TRANS_EMPTY | SSR_TRANS_END); // clear flags
21   S_CR|=SCR_TRANSMIT | SCR_TX_IRQ | SCR_TE_IRQ; // enable transmit & irqs
22
23   wait_event(write_complete,0);
24
25   // determine delay before next transmission
26   //
27   if (tx_state ==TX_IDLE)
28     tmp=LNP_WAIT_TXOK;
29   else
30     tmp=LNP_WAIT_COLL + ( ((unsigned char) 0x0f) &
31           ( ( unsigned char) len)+
32             ((unsigned char*)buf)[len-1]+
33             ((unsigned char) sys_time)     ) );
34   allow_tx=sys_time+tmp;
35
36
37 #ifdef CONF_TM
38   sem_post(&tx_sem);
39 #endif
40
41   return tx_state ;
42 }
43
44 #endif // CONF_LNP
```

Funktionen benytter følgende argumenter:

- `buf`, der angiver, hvor byteblokken, som skal sendes begynder
- `size_t len` angiver, hvor lang/stor blokken er.

Selve transmissionen benytter 3 pointere

- `tx_verify` peger på den byte, der skal kontrolleres næste gang.
- `tx_ptr` peger på den byte, som skal transmitteres næste gang.
- `tx_end` peger på den byte efter den sidste, der skal transmitteres.

Fra starten sættes `tx_verify` og `tx_prt` til at pege på `buf`. `tx_end` sættes til at pege på byten `buf` + `len`, som er byten efter den sidste, der skal sendes.

Kort sagt fungerer funktionen ved, at der sendes en byte til **SCI** (som sørger for at sende videre til IR-porten).

Funktionen vil ikke returnere, før alt er sendt, eller der er opstået en fejl. Herefter retuneres en værdi, som er 0, hvis overførelsen skete uden fejl. Alle andre værdier betyder fejl.

Modtagelsen

Vi går i nedenstående eksempel ud fra, at den fysiske transmission er succesfuld, og pakken udløser et interrupt på RCX'en. Dette interrupt bliver håndteret af funktionen `static void rx_handler(void)`(fra *lnp-logical.c*):

```
1
2  #ifdef  CONF_RCX_COMPILER
3  static  void rx_handler(void) {
4  #else
5  HANDLER_WRAPPER("rx_handler","rx_core");
6  void rx_core(void) {
7  #endif
8    lnp_timeout_reset ();
9    if (tx_state <TX_ACTIVE) {
10     // foreign bytes
11     //
12     allow_tx=sys_time+LNP_BYTE_SAFE;
13     lnp_integrity_byte (S_RDR);
14   } else {
15     // echos of own bytes -> collision detection
16     //
17     if (S_RDR!=*tx_verify) {
18       txend_handler();
19       tx_state =TX_COLL;
```

```
20        } else if ( tx_end <= ++tx_verify ) {
21          // let transmission end handler handle things
22          //
23          tx_state =TX_IDLE;
24        }
25      }
26
27      // suppress volatile modifier to generate bit instruction .
28      //
29      *(( char*) &S_SR) &=~SSR_RECV_FULL;
30    }
```

Hvis `tx_state<TX_ACTIVE` er sand, er den i modtagetilstanden, og sender bytes videre til `lnp_integrity_byte()`(se efterfølgende beskrivelse). Udover dette checker den også for kollisioner (se afsnit 2.3.5 på side 44)

```
1   void lnp_integrity_byte (unsigned char b) {
2     static unsigned char buffer [256+3];
3     static int bytesRead,endOfData;
4
5     if (lnp_integrity_state ==LNPwaitHeader)
6       bytesRead=0;
7
8     buffer [bytesRead++]=b;
9
10    switch(lnp_integrity_state ) {
11      case LNPwaitHeader:
12        // valid headers are 0xf0 .. 0 xf7
13        //
14        if (( b & (unsigned char) 0xf8) == ( unsigned char) 0xf0) {
15  #ifdef CONF_VIS
16      if ( lnp_logical_range_is_far ()) {
17              dlcd_show(LCD_IR_UPPER);
18        dlcd_show(LCD_IR_LOWER);
19      } else {
20        dlcd_hide(LCD_IR_UPPER);
21        dlcd_show(LCD_IR_LOWER);
22            }
23  #ifndef CONF_LCD_REFRESH
24      lcd_refresh ();
25  #endif
26  #endif
27            lnp_integrity_state ++;
28        }
29        break;
30
31      case LNPwaitLength:
32        endOfData=b+2;
33        lnp_integrity_state ++;
34        break;
35
36      case LNPwaitData:
37        if (bytesRead==endOfData)
38      lnp_integrity_state ++;
39        break;
40
```

```
41        case LNPwaitCRC:
42          if (b==(unsigned char) lnp_checksum(buffer, endOfData))
43       lnp_receive_packet (buffer );
44          lnp_integrity_reset ();
45      }
46    }
47    .
48    .
49    .
```

`lnp_integrity_byte()` tjekker, ved brug af en tilstandsmaskine, om **frames**/bytes'ene er gyldige. Den kontrollerer om den faktiske længde af dataen stemmer overens med den angivne længde, og den kontrollerer, at checksummen er korrekt. Såfremt at **framen**/byten er gyldig, bliver den sendt videre til `lnp_receive_packet()`:

```
1   void lnp_receive_packet (const unsigned char *data) {
2     unsigned char header=*(data++);
3     unsigned char length =*(data++);
4
5     // only handle non-degenerate packets in boot protocol 0xf0
6     //
7     switch(header) {
8       case 0xf0:          // raw integrity layer packet, no addressing.
9         if (lnp_integrity_handler )
10          lnp_integrity_handler (data, length );
11        break;
12
13      case 0xf1:          // addressing layer.
14        if (length >2) {
15          unsigned char dest =*(data++);
16
17          if (LNP_HOSTADDR == (dest & LNP_HOSTMASK)) {
18            unsigned char port =dest & LNP_PORTMASK;
19
20            if (lnp_addressing_handler [port ]) {
21              unsigned char src =*(data++);
22              lnp_addressing_handler [port ]( data, length -2,src);
23            }
24          }
25        }
26
27    } // switch(header)
28  }
```

Funktionen starter med at kigge på det første felt i headeren på den pakke, den har fået med som parameter. Hvis det er en integritetspakke vil den have værdien 0xF0 i headerfeltet. Hvis der står 0xF1 er det en adresseringspakke.

Hvis den får en integritetspakke, kontrollerer funktionen, om der er opsat en handler til at tage imod pakken, og i så fald, sender den pakken videre til denne. Hvis det drejer sig om en adresseringspakke, kontrollerer funktionen, hvorvidt pakken

er havnet det rigtige sted, og hvis der er sat en handler op på den rigtige port, sender den pakken videre til denne. Hvis der ikke er nogen handler til at håndtere pakken, bliver den ignoreret. Brugeren skal selv implementere en **packet handler**, hvilket næste afsnit vil omhandle.

Packet handlers

For at modtage data til et brugerprogram, skal der opsættes **packet handlers** til at håndtere dette. Hvis der ankommer data til en port, som håndteres af en **packet handler** kaldes denne.

Det er op til brugeren selv at implementere **packet handleren** - LNP sørger kun for, at den bliver kaldt, når der ankommer data på den pågældende port. Packet handleren skal have følgende interface:

```
1 void min_handler(const unsigned char *data, unsigned
2 char length, unsigned char src)
```

`data` er en pointer til det data, som er modtaget på porten, `length` er størrelsen på datapakken, og `src` er adressen på det program, som sendte datapakken. For at installere en handler skal følgende funktion kaldes:

```
1 lnp_addressing_set_handler (MIN_PORT, min_handler)
```

Denne funktion vil installere `min_handler` på porten `MIN_PORT`.

Der skal installeres packet handlere på alle de porte, hvor der forventes at ankomme data. Ovenstående to eksempler viser, hvordan handleren virker på adresseringspakker. Pakker fra integritetslaget virker på sammse måde, med den undtagelse, at der kun installeres en handler, da der ikke er mulighed for at sende til specifikke porte.

2.3.7 Brug af LNP

Når LegOS køres på en RCX, er LNP altid aktivt, så det er ikke nødvendigt at initialisere den selv. Hvis der skal bruges LNP på en PC, initialiseres den med `lnp_init(tcp_hostname,tcp_port, lnp_address, lnp_mask, flags)`. 0 kan bruges som default for alle parametre. `tcp_hostname` og `tcp_port` er vært og port på den **lnpd daemon**, som der oprettes forbindelse til.

`lnp_address` er netværksadressen for pågældende RCX på LNP netværket. `lnp_mask` bestemmer (udfra den 8-bit LNP adresse), hvilke bits, der sætter netværksenheden, og hvilke bits, der sætter port nummeret. `lnp_init` vil returnere

0, hvis initialiseringen blev korrekt udført.

2.3.8 Afsluttende overvejelser

Efter at have analyseret LNP's protokolstak har vi gjort os nogle overvejelser omkring opbygningen af denne. Som det ses på figur 2.11 på side 38, ligger de to øverste lag, integritetslaget og adresseringslaget, side om side. Ifølge den klassiske opbygning af en protokolstak ([Tan96]) skal lagene ligge fra top til bund, og tilbyde veldefinerede tjenester/interfaces til hinanden. I LNP er adskillelsen af lagene ikke særlig skarp.

Dette er endnu mere tydeligt på modtagersiden; hvor det logiske lag validerer de enkelte bytes/**frames** ved hjælp af en en tilstandsmaskine fra integritetslaget (`integrity_byte()`). Først herefter sender den pakken videre til `lnp_receive_packet()`, som afgør, om den skal op til integritetslaget eller adresseringslaget.

Dog er det vigtigt at huske på, at da Marcus L. Noga designede og implementerede protokolstakken til LegOS, har han sandsynligvis lagt stor vægt på at gøre den så lille og kompakt som muligt. Dette ses også i selve opbygningen af koden, som bruger masser af indlejrede funktioner og er forholdsvis svær at læse. Men derved bliver der jo mere plads til brugerprogrammer, og formålet med LegOS er jo at kunne lave spændende brugerprogrammer.

2.4 Netværk på RCX - de fysiske rammer

I dette afsnit vil vi forsøge at klarlægge de muligheder, der er for at få to eller flere RCX'er til at kommunikere sammen. Først vil vi se på, hvilke muligheder, der er for at forbinde RCX'erne, med fokus på porte og forbindelsen mellem dem. Dernæst vil vi analysere to forskellige netværksarkitekturer, for at se på mulighederne for at forbinde mere end to RCX'er med hinanden.

2.4.1 Forbindelsen

For at få to computere til at kommunikere, kræves der en eller anden form for forbindelse mellem de to. Dette gælder selvfølgelig også for RCX'erne. Vi har diskuteret en række muligheder for at forbinde de to enheder, og vi vil gennemgå de vigtigste her. Endvidere vil vi beskrive fordele og ulemper ved de forskellige metoder, samt vurdere, hvilken eller hvilke, vi skal fokusere på fremover.

IR-porten

RCX klodsen har en indbygget infrarød port, som bruges til at uploade programmer fra en PC. IR-porten er oplagt til kommunikation mellem to RCX'er og LNP bruger denne. LNP er beskrevet i 2.3.2. Upload hastigheden kan foregå ved 2 hastigheder, nemlig ved enten 4800 baud eller 2400 baud.

IR-porten har den åbenlyse fordel at kommunikationen er trådløs, samt at den benytter **SCI**. Dette åbner nogle spændende muligheder med hensyn til, hvad netværket kan bruges til. Det kunne for eksempel være muligt, at der bruges en RCX til at fjernstyre en anden.

Fra inputport til inputport

En lyssensors virkemåde har givet os idéen til denne måde at forbinde to RCX'er på (se figur 2.17, side 57). Tanken er her, at inputporten både kan sende og modtage, og at vi derfor kan nøjes med at bruge en port på RCX'en for at lave et netværk. Der opstår dog nogle problemstillinger med denne fremgangsmåde, og disse vil vi komme ind på nedenfor. En lyssensor til RCX'en indeholder en lysdiode, en kondensator og én lysfølsom modstand. Når RCX'en bruger en lyssensor på sin inputport, foregår det ved, at porten først sender noget strøm til kondensatoren i sensoren. Herefter aflades kondesatoren ved at forbinde den til den indbyggede

Figur 2.17: To RCX'ere i netværk

lysdiode. Mens afladningen foregår, står porten og måler på den lysfølsomme diode[14].

Ovenstående udredning af en lyssensors virkemåde fortæller os, at en inputport både kan sende en strøm ud og måle på porten. Altså kan vi bruge inputporten til både at sende og modtage. Kommunikationen kan dog kun foregå i halv duplex, da porten ikke kan sende og modtage på samme tid.

Med dette mener vi, at porten ikke kan sende og modtage parallelt. Den kan derimod godt sende og modtage samtidigt. Med samtidigt menes, at porten skifter mellem at stå til rådighed for sender og modtager processen. Når vi i det følgende snakker om at sende og modtage på én gang, mener vi samtidigt og ikke parallelt.

For at forstå inputporten, satte vi et oscilloskop til at måle på den. Vi fandt ud af at, porten med faste mellemrum sender strøm ud til de aktive sensorer. Dette var ikke nogen stor overraskelse, da vi vidste at, for eksempel en lyssensor skal bruge strøm for at fungere. Det overraskende var i stedet, at vi ikke kunnne slukke for denne strøm.

Sensorportene er koblet til port 6 på Hitachi chippen og vi forventede derfor at kunne afbryde strømmen på porten, ved at sætte de rette bits i det register der styrer port 6 (**P6DDR**). Under vores tests viste det sig dog, at dette ikke var muligt. Vi er ikke sikre på, om det er LegOS, der holder porten i gang, eller om det er et

[14]http://www.crynwr.com/lego-robotics/light-sensor.htm

hardwareproblem.

Det kan ikke lade sig gøre at sende en bitstrøm fra inputporten. Hvis porten aktiveres, mens man ikke sender data, kan den strøm, som sendes opfattes som data af modtageren. Om ikke umuliggør, så besværliggør dette implementationen i en betydelig grad, hvilket er grunden til at vi ser bort fra denne mulighed.

Fra outputport til inputport

Idéen her er, at vi bruger en port til at sende og en port til at modtage. Ulempen ved denne fremgangsmåde er, at vi bruger to porte på hver RCX til netværk, og dermed er der en port mindre til rådighed for brugerprogrammer. Figur 2.18 viser opstillingen med to kabler. Med en port til modtagelse og en til afsending af data. For at teste, hvorvidt denne opstilling kan anvendes, har vi skrevet et par simple

Figur 2.18: RCX'er forbundet med to kabler.

testprogrammer. Det ene bruger LegOS' motorportdrive r(`motor_a_dir()`) til skiftevis at sætte outputporten til høj og lav. Det simple bitmønster 010101010.

Det andet program bruger `ds_scale()`, en funktion fra LegOS, til at læse på inputporten. Med disse simple programmer downloadet til hver sin RCX, kunne vi konstatere, at det er muligt at sende data fra en outputport til en inputport.

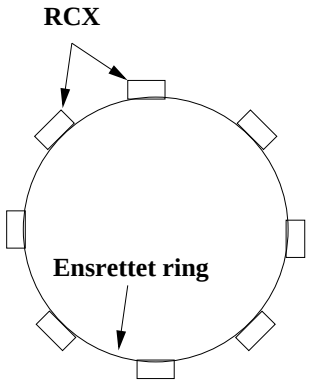
2.4.2 Netværks topologier

Afsnittet her beskriver to topologier til den fysiske implementering af et netværk. Disse to er **Token Ring netværk** og **Stjernenetværk**. Vi vil undersøge fordele og ulemper, for at danne basis for en senere beslutning, om hvilken arkitektur, vi

ønsker at benytte.

Token ring netværk

Da erfaringer viser, at en enkelt RCX ikke kan holde til at være på ledningsbaseret netværk med mere end to til tre andre RCX'er, uden at portene brænder sammen, kunne vi bruge et simpelt **Token ring netværk** [Tan96] til at forbinde dem med. Alle maskinerne skal serieforbindes, og de skal sende et såkaldt **token** rundt på netværket. Dette **token** skal repræsenteres som et specielt bitmønster. Ydermere

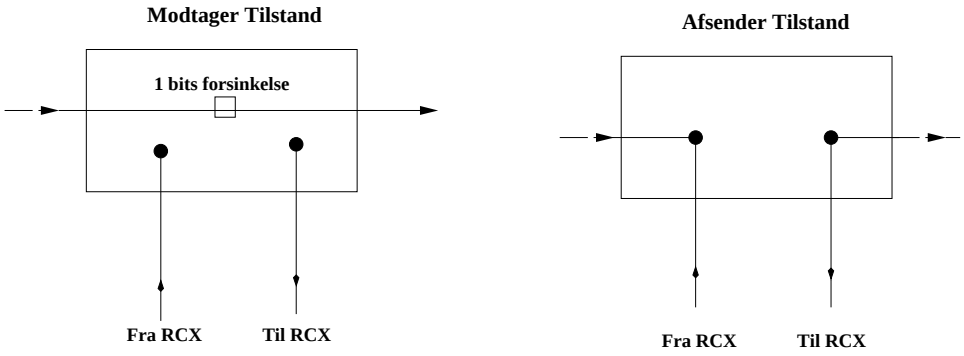


Figur 2.19: Her ses strukturen i et **token ring netværk**

kan en RCX være i to tilstande: Lytte eller sende tilstand. Hvis den lytter, skal den bare sende **token**'et (og alt andet data) videre til næste maskine. Dette skal ske ved, at hver gang den modtager en bit, kopieres denne bit ind i en buffer, og sendes derfra videre til næste RCX på netværket. Fra denne buffer har RCX'en mulighed for at aflæse og ændre bit'en. Denne kopiering medfører mindst et 1-bit delay ved hver maskine (se figur 2.20, side 60).

Når RCX'en skal sende noget ud på netværket, skal den "gribe" **token**'et (hvis muligt), ved at ændre den første bit. Den må endvidere kun sende i et bestemt stykke tid, **token holding time**, for ikke at monopolisere netværket.

Når de bits som afsenderen sender ud på netværket, når hele ringen rundt, og kommer tilbage til den selv, skal den fjerne dem. Når RCX'en har fjernet den sidste bit fra det data, den har sendt ud, skal den sende et nyt **token** ud på netværket. Hvis ingen maskiner har noget at sende, vil **token**'et cirkulere hurtigt rundt, og derfor skal den forsinkelse, som hver maskine på netværket giver, være stor nok til, at et helt **token** kan være i ringen. Hvis man gerne vil have, at modtageren skal bekræfte de modtagne pakker den får, kan den ikke vente på, at den får **token**'et - det vil tage alt for lang tid. Derfor kan modtageren (sædvanligvis efter at have kontrol-



Figur 2.20: Her ses, hvordan en maskines modtage/afsendetilstand rent begrebsmæssigt håndterer datastrømmen på.

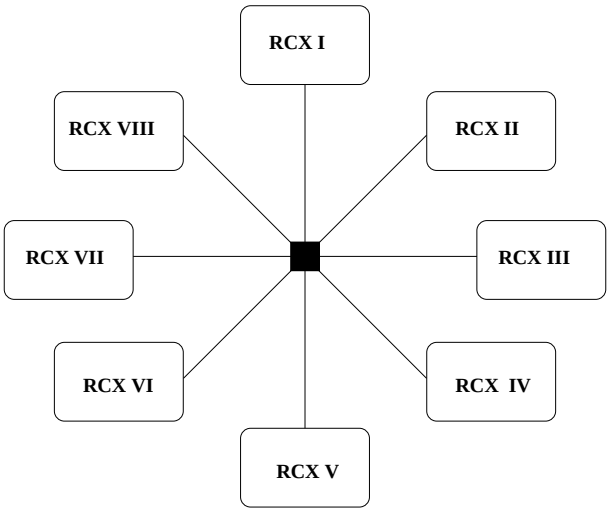
leret checksummen) ændre en kontrolbit til sidst i pakkens header, lige inden den sender pakken tilbage til afsenderen. Derved sender den en bekræftelse tilbage til afsenderen, på at pakken blev accepteret. Hvis den ikke sætter denne sidste bit, ved afsenderen, at pakken ikke blev accepteret, og kan eventuelt retransmittere pakken. Da den sidste bit ikke er inkluderet i checksummen, er den ofte repræsenteret to gange, for at øge pålideligheden.

Hvis en af RCX'erne mister kontakten til netværket, ved eksempelvist et strømsvigt, vil hele netværket gå ned på grund af netværksarkitekturen, og de hardwaremæssige muligheder som RCX'en tilbyder. Havde det været et almindeligt PC-netværk, ville der typisk være et specielt netværks-/interfacekort til **Token ring netværket**, som kunne sørge for at sende bits'ene videre, hvis dens maskine eksempelvist var gået ned.

Stjernenetværk

Idéen med et **stjernenetværk** [15] er, at alle hosts er koblet sammen centralt, som vist på figur 2.21 på side 61. Alle klienterne er forbundet 1:1 til eksempelvist en **hub**/**switch**. Et karakteristika ved et sådant netværk er, at alle de klienter, som er tilsluttet netværket, kan sende, når de vil. Derfor opstår der også flere kollisioner, se mere om kollisionsdetektering i fejlhåndtering (afsnit 2.3.5, side 44). Der er også mulighed for at bruge **fuld duplex**, det vil sige at kunne sende og modtage på samme tid parallelt. Det er også muligt at koble flere stjernenetværk sammen, og hvert enkelt af disse bliver så til et segment. Hvis forbindelsen mellem segmenterne bryder sammen, opstår to selvstændige netværk, og **hosts**'ene på hvert enkelt segment, vil stadig kunne kommunikere.

[15] http://www.ibilce.unesp.br/courseware/networks/pt2_2.htm



Figur 2.21: Et **stjernenetværk** med otte RCX'er

En fordel er, at det er nemt at tilføje/fjerne hosts på netværket.

3 Design

Indholdsfortegnelse

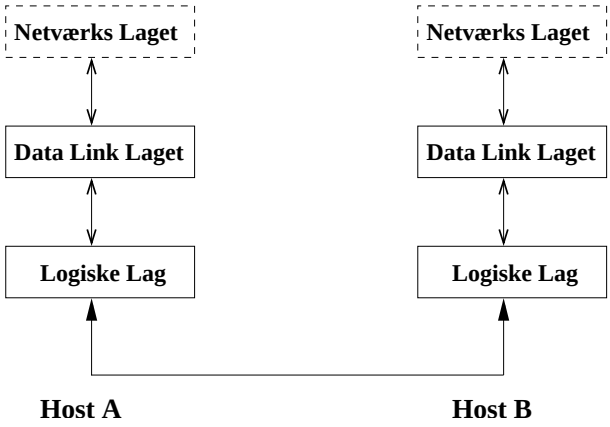
Dette kapitel omhandler vores tanker og idéer omkring design af en ny netværksprotokol til LegOS. Kapitlet forklarer, hvordan vi har tænkt os at opfylde den systemdefinition(se side 4), vi tidligere har fremlagt. Vi vil præsentere de designvalg, vi har truffet, og argumentere for de enkelte valg.

I dette kapitel vil vi også se på interfacet mellem lagene, samt give en beskrivelse af det interface der skal gøre det muligt for andre at benytte vores netværksprotokol.

Vi laver en helt ny protokolstak, som skal indeholde de lag, der ses på figur 3.1. Kapitlet indeholder et afsnit om hvert lag i stakken. Navnene på de forskellige lag er inspirerede af OSI-modellen og LNP (se afsnit 2.3.2 side 37), og det er med disse som forbilleder, vi vil implementere vores netværksprotokol.

Efter en analyse af OSI-modellen og LNP, har vi grundlaget til at designe protokolstakken. Vi har taget de forskellige krav fra vores systemdefinition, og knyttet dem til et lag i protokolstakken udfra de specifikationer, som gives i OSI-modellen. OSI-modellen har udelukkende været brugt som inspiration til navngivning af lagene, samt placering af funktionalitet. Analysen af LNP har givet os et konkret eksempel at kigge på.

Valget af en lagdelt arkitektur kan begrundes med, at den giver mulighed for klart at adskille de forskellige lag, og hermed de forskellige funktioner lagene indeholder.



Figur 3.1: Vores protokolstak.

I henhold til overvejelserne i afsnit 2.4 på side 56, har vi besluttet at forbinde RCX'erne ved hjælp af to porte/kabler. Figur 2.18 på side 58 viser dette.

3.1 Det logiske lag

Det nederste lag i vores protokolstak kalder vi for det logiske lag, efter inspiration fra LNP. Dette lag har til opgave at bruge I/O porte på RCX'en til at sende og modtage en bitstrøm. For at realisere dette udvikler vi en device driver til at styre I/O portene. Endvidere stiller det logiske lag en række funktioner til rådighed for datalink laget i et veldefineret interface

3.1.1 Device drivere

Til at styre porten skal som nævnt bruges en device driver. I LegOS er der allerede implementeret drivere til at bruge portene[1]. Disse drivere er skrevet specifikt til brug af for eksempel motorer eller lyssensorer, hvilket bevirker, at de ikke vil være optimale for os.

Istedet skriver vi en helt ny device driver, som er i stand til at portene på en måde, så vi kan bruge dem til at sende data. Dette er en af det logiske lags vigtigste funktioner, da det forbinder hardwaren med protokolstakken.

3.1.2 Repræsentation af bits

En outputport på RCX'en kan sende spænding ud i to retninger, det vil sige at den kan vende polerne. Dette bruges f.eks. til at drive en motor begge retninger. Endvidere kan porten selvfølgelig også være slukket.

Til at repræsentere en binær bit bruger vi **Manchester Encoding**(se afsnit 2.3.4 på side 43). Grunden til at vi bruger dette er, at det giver os en måde, hvorpå vi kan adskille **frames**. Når en **frame** slutter, sender vi tre gange høj efterfulgt af tre gange lav, et bitmønster, som aldrig vil forekomme i de aktuelle data, når vi bruger **Manchester Encoding**. Dette gør, at modtageren ikke kan være i tvivl om, at den modtagne **frame** er slut. Endvidere sørger **Manchester Encoding** også for, at vi kan skelne mellem ingen aktivitet fra senderen og sendingen af en 0 bit. På figur 2.14 på side 43 ses hvordan vi representerer den binære streng 00110011 ved hjælp af **Manchester Encoding**.

Der opstår en problemstilling omkring timing af afsendelse af bitstrømmen. Det er et spørgsmål om, hvor lang tid strømmen skal være slået til og fra, for at sende en bit. Ligeledes skal vi finde ud af, hvordan vi sørger for, at sender og modtager er synkroniserede. Disse spørgsmål vil vi få svar på, ved hjælp af en række tests, som er beskrevet i sektion 4.2.4 på side 83.

[1] Se sektion 2.2.5 for en beskrivelse af disse.

3.1.3 Interface

For at adskille lagene i stakken, er det vigtigt med veldefinerede interfaces mellem de individuelle lag. Vi har defineret interfacet mellem det logiske lag og datalink laget således:

- **Logisk -> Datalink :** Funktionen `int getByte(unsigned char *)` i det logiske lag bliver kaldt fra datalink laget. Funktionen tager en pointer til en `char` som argument, og denne `char` fyldes med modtagne databits. Funktionen returnerer 1 hvis det gik godt, og 0 hvis det mislykkedes.

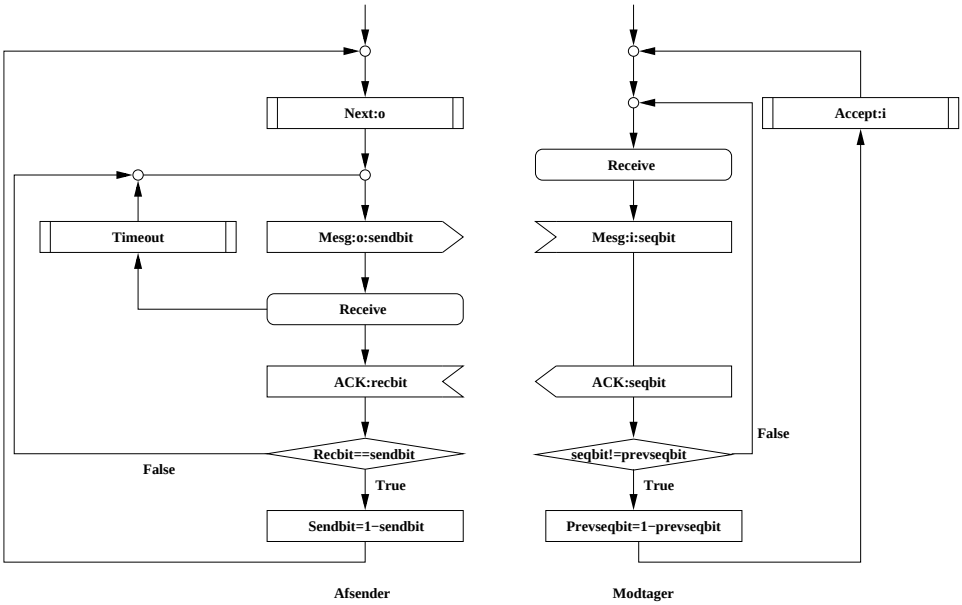
- **Datalink -> Logisk :** Funktionen `int write2Buffer(unsigned char *, int)` funktionen i det logiske lag bliver kaldt fra datalink laget. Funktionen tager som argument en pointer til et bytearray, indeholdende de data der skal sendes. Antallet af dataelementer er det andet argument.

3.2 Datalink laget

Datalink laget har til opgave at stille data fra det logiske lag til rådighed for enten et brugerprogram eller et overliggende protokollag. Det data som datalink laget giver videre, skal være fejlfri og de enkelte pakker skal komme i den rigtige rækkefølge. Til at håndtere datalinkpakker bruger vi en **alternating bit** protokol, som er en forgænger til **sliding window**[2] protokol, hvor både sende- og modtagevinduet har størrelsen 1.

Figur 3.2 viser et flowdiagram af udførelsen af den designede **alternating bit protokol**. På figuren er `o` den pakke som senderen vil afsende. Modtageren modtager `o`, gemmer den i `i` og sender et **ACK** tilbage med nummeret `seqBit`. Senderen sammenligner dette med `sendBit`, hvis de er ens, er det et **ACK** på den rigtige pakke og senderen sætter `sendBit` til det modsatte af dens værdi. Hvis ikke de er ens var det et **ACK** på en forkert pakke og `o` gentransmitteres. Efter at modtageren har sendt **ACK** på `o`, tjekker den om pakkens sekvensnummer(`seqBit`) er ulig med det fra den foregående pakke, hvis dette er tilfældet accepteres pakken. Ellers droppes pakken og modtageren venter på en ny pakke. Figuren viser også hvordan senderen vil gentransmittere en pakke, hvis ikke den for et **ACK** inden **timeout**.

[2] For at læse mere om **Slinding Window** og **alterating bit**, se afsnit 2.3.3 på side 40 og [Tan96] kapitel 3



Figur 3.2: Alternating bit protokol med timeouts

3.2.1 Datalinkpakken

Protokolstakkens primære datastruktur er datalinkpakken som ses på figur 3.3. Den første byte i headeren har den udformning som tabel 3.1 viser.

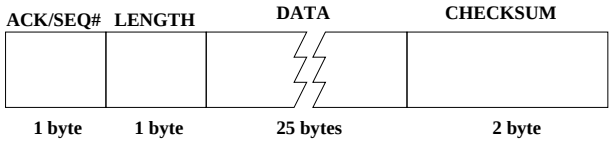
ackFlag	piggybackFlag	ubrugt	ubrugt	ubrugt	ubrugt	seqBit	ackBit

Tabel 3.1: Den første byte i datalinkpakken.

Hvis **ackFlag** bit'en er sat, betyder det at der er **ACK** information i **ackBit**'en som skal læses. **seqBit** indeholder sekvensnummeret. Hvis **piggybackFlag** er sat, betyder det, at pakken både indeholder data og **ACK** information.

Pakkens anden byte er karakterantallet eller mængden af data (**LENGTH**). De næste 25 bytes bruges til data (**DATA**), og de sidste to bytes er en 16-bit checksum (**CHECKSUM**).

En sådan pakke bygges op af funktionen `fillSendBuffer()` og modtageren deler den ind igen med funktionen `getData()`. Begge disse funktioner kommer vi nærmere ind på i afsnit 3.3.



Figur 3.3: En datapakke som den ser ud på datalink laget.

3.2.2 Fejhåndtering

Udfra vores analyse af de to typer fejlhåndtering, henholdsvis fejldetektering og fejlkorrigering, har vi valgt at bruge fejldetektering og retransmittering. Vi har valgt at gøre dette udfra det synspunkt, at det er simplere at implementere og at overhead'et ved at bruge fejlkorrigering er for stort. Se sektion 2.3.5 for en udredning af dette.

Datalink laget udregner en checksum, som det sætter på hver udgående pakke. Når modtageren får en pakke, udregner den checksummen for at kontrollere, om den får den samme værdi, som da den blev udregnet hos senderen. Hvis dette ikke er tilfældet, droppes pakken og modtageren sender ikke en **ACK** tilbage. Dette resulterer i, at pakken bliver gentransmitteret, når senderens timer for den pågældende pakke løber ud. Af samme grund bliver pakken også gentransmitteret, hvis ikke den når frem til modtageren. Denne form for fejlhåndtering gør, at vi kan garantere, at pakker når fejlfrit frem til modtageren.

Til beregning af checksummen benytter vi os af den samme algoritme som, LNP bruger. Algoritmen er beskrevet og analyseret i sektion 2.3.5.

Når datalink laget modtager en pakke, kontrollerer den om sekvensbit'en er det modsatte af det den var på den foregående pakke. Hvis dette ikke er tilfældet, må pakken være gået tabt. Dette håndterer datalink laget, ved ikke at sende en **ACK** tilbage, hvilket resulterer i at senderen retransmitterer pakken. Senderen vil dog kun retransmittere pakken tre gange. Hvis ikke den får en **ACK** tilbage efter tredje forsøg, betragtes modtageren som unåelig, og den opgiver at sende pakken. Hvis dette sker, sender datalink laget en fejlbesked op til brugerprogrammet.

På modtagersiden er der fejlhåndtering i form af et timeout, hvis der modtaget for lidt data. Betragt følgende scenario: modtageren får det bitmønster som indikerer at en **frame** er slut, inden den har modtaget data nok til at det bliver til en hel pakke. Dette kan kun ske, hvis støj på linien eller andet har ødelagt datapakken.

For at undgå at komme til at vente uendeligt på data der ikke kommer, bruger vi en timer til at sikre os, at vi bryder ud af `getData` efter en given tid.

3.2.3 Interface

Det interface datalink laget stiller til rådighed for det overliggende brugerprogram er defineret således:

- **Datalink laget -> Applikations niveau**
  Funktionen `getPacket(unsigned char *)` ligger også i datalink laget. Funktionen tager som parameter en pointer til et array, som skal fyldes op med modtagene data. `getPacket()` returnerer mængden af data, i antallet af bytes.

- **Applikations niveau -> Datalink laget**
  Funktionen `int fillSendBuffer(unsigned char *, unsigned char)` tager som parameter en pointer til et array med data, som kalderen ønsker skal sendes. Den anden parameter er længden på dataarrayet. Funktionen returnerer 1 hvis afsendelsen lykkedes og 0 hvis ikke. Dette er ikke en garanti for at pakken modtages, kun at den er blevet givet til det logiske lag, som har sendt den afsted.

3.3 Processer og funktionalitet

3.3.1 Processer

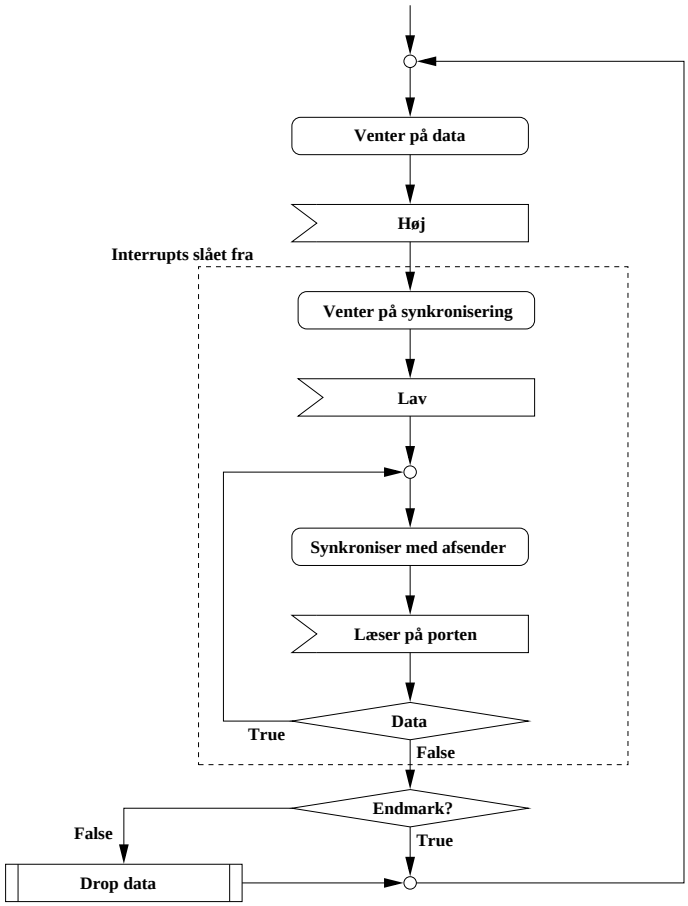
For at starte vores netværksprotokol, skal der laves 3 nye processer som **task manageren** skal begynde at afvikle. En ny proces startes med funktionen `execi()`, som blandt andet tager en pointer til startadressen på det kode, som skal udføres.

De tre processer bliver startet af en funktion, som bliver kaldt fra *kmain.c*, som kaldes fra **ROM**'en for at starte kernen.

Det logiske lags modtageproces

Denne proces har til ansvar at holde øje med, om der kommer noget data på inputporten. En tråd i processen læser hele tiden på A/D konverterens dataregister, for at se om der er indkomne data på inputporten. Til dette bruges funktionen `readInput()`, som returnerer 0 eller 1 afhængig af den værdi den aflæser fra A/D konverterens dataregister.

Figur 3.4 viser et flowchart over den proces hvori funktionen der modtager data



Figur 3.4: Flowchart over det logiske lags modtagedel.

på det logiske lag kører. Processen er i **waiting**[3] tilstanden indtil den læser høj på inputporten, hvorefter den går ind i den region hvor **interrupts** er slået fra. Her går den i **waiting** tilstanden igen, indtil der læses lav på porten. Efter dette følger en synkronisering med afsenderen, hvorefter der læses data ind fra inputporten. Dette vil fortsætte indtil der ikke kommer mere input på porten. Når der ikke kommer mere, kontrolleres det om der kom et **Endmark**, hvis dette er tilfældet vil **framen** blive accepteret ellers droppes den.

Istedet for at bruge **busy waiting** bruger vi en funktion i LegOS som hedder `wait_event()`. Vi giver `wait_event()` en pointer til `readInput()`. Når denne returnerer 1 betyder det, at der måles høj på porten. Dette fortæller modtageren, at senderen har noget data at sende. Nu fortæller `wait_event()` scheduleren, at denne proces er **sleeping**. Så længe `wait_event()` venter på at

[3] I afsnit 2.2.3 på side 24 er procestilstandene i LegOS beskrevet.

`readInput()` returnerer 1, er processen i tilstanden **waiting**.

Herefter eksekverer tråden en funktion, der skal modtage de aktuelle data. Som beskrevet ovenfor bliver **interrupts** slået fra, for at undgå at scheduleren skifter processen ud til fordel for en anden proces med højere prioritet, hvilket vil bringe sender og modtager ud af synkronisering.

Nu går overførslen af den egentlige data igang. Der læses i A/D konverterens dataregister, og afhængigt af værdien heri, skrives en 1 bit eller en 0 bit i `readBuffer`. Tre gange høj betyder, at denne **frame** er slut. Når dette bitmønster modtages, er overførslen, slut og modtagetråden slår **interrupts** til igen, hvorefter den terminerer.

Datalinklagets modtagerproces

Denne proces har til ansvar at skille en datalinkpakke ad, tjekke at pakken har det rigtige sekvensnummer, udregne om checksummen er rigtig og sende det data, som pakken indeholder videre til et brugerprogram eller et højere protokolstakniveau.

Funktionen `getData()` bruger `wait_event()`, så den står i **waiting** tilstanden, indtil der kommer data fra det logiske lag. Når dette sker signalerer `wait_event()`, at processen er **sleeping**. `getData` kalder funktionen `getByte()`, som ligger i det logiske lag, til at hente en enkelt byte op fra det logiske lag.

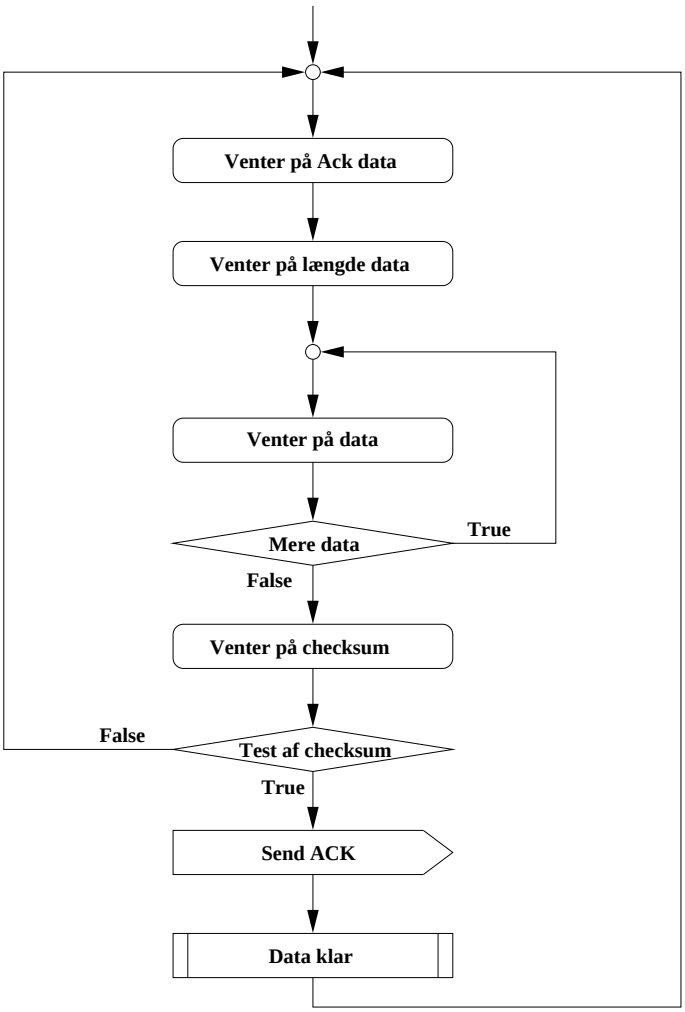
Figur 3.5 viser hvordan datalink laget modtager en datalink pakke. Det første der læses er **ACK** feltet, herefter **LENGTH** feltet. Dernæst læses der data op til **LENGTH**. Hvis checksummen ikke fejler accepteres pakken og der afsendes en **ACK** på pakken.

Det logiske lags sendeproces

Denne proces har til opgave at tage data fra datalink laget, og sende det ved hjælp af den device driver der styrer outputporten. I denne proces bruger vi også funktionen `wait_event()`. Her venter vi på, at datalink laget fylder `writeBuffer` op med data. Når denne er fyldt går processen ind i sin sendetilstand, hvor **interrupts** er slået fra - igen for at sikre, at afvikleren ikke skifter processen ud.

Det første der sker i sendetilstanden er, at processen sender høj i `x` millisekunder, for at fortælle modtageren, at der er data på vej. Efter perioden `x` sendes lav, hvorefter processen er klar til at sende det egentlige data.

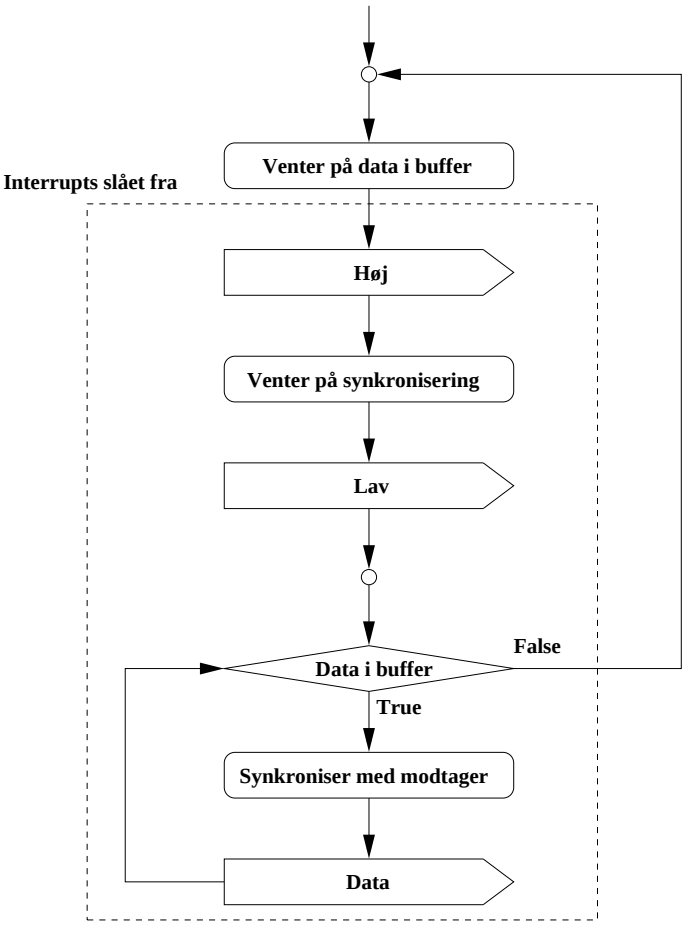
Afsendelsen af data foregår på den måde, at processen starter en tråd med funktio-



Figur 3.5: Flowchart over datalink lagets modtagedel.

nen `write2Buffer()`. `write2Buffer()` sender hver enkelt bit, ved hjælp af device driveren. Inden tråden terminerer slåes **interrupts** til igen, og processen går ud af sendetilstanden. Figur 3.6 viser dette forløb.

De tre processer der er beskrevet ovenfor, står for afsendelse af data på det logiske lag, modtagelse af data på det logiske lag, samt modtagelse af data på datalink laget. For at sende data fra datalink laget skal funktionen `fillSendBuffer()` kaldes fra et højere niveau. Dette kunne være et brugerprogram eller et højere niveau i protokolstakken.



Figur 3.6: Flowchart over det logiske lags sendedel.

3.3.2 Funktionalitet

fillSendBuffer(): Denne funktion tager som parameter en pointer til et array af bytes, som er det data der skal sendes. Funktionen har til opgave at opbygge en datalinkpakke og fylde data i den. Pakken opbygges som på figur 3.3.

Inden funktionen begynder at opbygge en pakke, sætter den et flag, som skal være med til at afgøre, om en **ACK** skal **piggybackes** på den udgående datapakke. Når pakken er opbygget, sendes den til det logiske lag, ved hjælp af funktionen `send2Logisk()`, som kalder `write2Buffer()` fra det logiske lag.

4 Implementation

Indholdsfortegnelse

Dette afsnit omhandler implementeringen af protokolstakken. Afsnittet vil kommentere på de ændringer vi har lavet i forhold til vores design. Endvidere vil vi give eksempler på, hvordan vi har implementeret forskellige elementer af netværket. Kapitlet vil også indeholde en opsummering af de problemstillinger, vi er løbet

ind i undervejs, og vi vil forklare hvordan vi har løst disse, eller hvorfor vi ikke fik løst dem.

Sidst i kapitlet vil vi præsentere en brugermanual, som har til hensigt at sætte eventuelle brugere istand til at benytte sig af vores netværksprotokol.

4.1 Status på implementationen

Dette afsnit har til formål at klarlægge, hvor meget af designet vi har nået at implementere. Endvidere vil vi komme ind på problemer, vi er stødt på under implementeringen, og vi vil give eksempler på hvordan vi har implementeret væsentlig funktionalitet. Afsnittet er delt op i to; et underafsnit om implementeringen af det logiske lag samt et om datalink laget.

4.1.1 Det logiske lag

Det logiske lag har til opgave at oversætte data til en bitstrøm og benytte hardwaren til at sende det til modtageren. Implementering af dette lag omhandler således brug af I/O enheder som inputporte og outputporte.

Implementeringen af det logiske lag tog meget længere tid end beregnet, og indebar flere problemer end først antaget. Et af vores største problemer var omkring timingen mellem sender og modtager. Dette og andre problemstillinger vil blive beskrevet i det følgende.

Device drivere

Vi har brug for at kunne styre portene på RCX'en på en måde, så vi kan bruge dem til at sende/modtage en bitstrøm. For at være i stand til dette, har vi skrevet tre simple device drivere.

Outputporten bruger vi til at sende med. For at implementere dette styrer vi porten med de nedenstående drivere, som henholdsvis tænder og slukker porten. Altså bruges den ene til at sende høj, og den anden lav.

```
1  void send_1(){
2    asm("
3      mov.w #0x40, r1
4      mov.w r1, @0xF000          ; set outputport A high
5    ");
```

```
6  }
```

```
1  void send_0(){
2    asm("
3      mov.w #0x00, r1
4      mov.w r1, @0xF000          ; set outputport A low
5    ");
6  }
```

Funktionerne virker ved at skrive til registret på adressen 0xF000. Registret styrer alle tre outputporte. Vi bruger kun port A, så det er kun dens tilstand driveren ændrer på.

Inputporten bruger vi til at modtage bits med, og vi styrer den med den nedenstående device driver.

```
1  long calcSens_new(){
2    int i = 0;
3    *adcsr = 0 x2A;              // start a/d conversion on inputport 1
4    for(i = 0; i < 4; i++);
5    while(*adcsr < 0 x80);       // wait for a/d conversion end
6    return (* datareg < 128);    // return 1 or 0
7  }
```

A/D konverteren startes ved at skrive til **ADCSR** registret på adresse 0xFFE8. Herefter ventes der fire iterationer i en for-løkke. Dette gøres for at undgå deadlock. Det er vores erfaring, at LegOS låser, hvis ikke denne venteperiode indføres efter start af A/D konverteren. Funktionen venter nu på, at A/D konverteren signalerer, at den er færdig med at læse på porten. Dette signaleres ved at A/D konverteren sætter **A/D End Flag**(**ADF**). A/D konverteren gemmer resultatet i to 8-bit registre **ADDRCH** og **ADDRCL**, som findes på adresserne 0xFFE4 og xFFE5. Dog bruges kun de to mest betydende bits i **ADDRCL** - dvs at resultatet er en 10-bit værdi. Når konverteringen er færdig, læser funktionen fra **ADDRCH** registret og returnerer 1 eller 0, hvis værdien i registret er henholdsvis større eller mindre end 128.

4.1.2 Datalink laget

Datalink lagets opgave er, at modtage en datamængde fra det logiske lag, og stille det til rådighed for brugerprogrammer som brugbar data. Samt at sende data fra et brugerprogram videre til det logiske lag. Med brugbar mener vi fejlfri data i den rækkefølge, som det blev sendt. Datalink laget implementerer en **alternating bit** protokol, som specificeret i vores design.

Modtagelse af en datalinkpakke

Når modtageren skal læse data fra det logiske lag, står den hele tiden og tjekker, om der er noget i bufferen `readBuffer[]`. Hvis dette er tilfældet, bliver det undersøgt af funktionen `getData()`. Denne funktion starter med at undersøge om bit 7 i den første byte er 1. I så fald er det en **ACK** pakke, og den henter **ACK** sekvens bitten fra bit 1. Denne bit bliver sendt videre til `ackHandler()` funktionen, som kontrollerer, at der **ACK**es på den rigtige sekvens bit.

Derefter undersøger funktionen om bit 6 er 1. Hvis den er, og bit 7 også var, ved funktionen, at det er en **piggybacked ACK** værdi. Den mindst betydende bit er pakkens sekvensnummer. Hvis bit 6 var 0, vil der ikke komme noget efterfølgende data, da det så er en ren **ACK** pakke.

Hvis der er data i pakken, henter den næste byte fra `readBuffer[]`. Denne byte indeholder antallet af dataelementer, som efterfølgende hentes. Den sidste og anden sidste byte indeholder tilsammen en 16 bit checksum.

```
1  void getData(){
2    int i = 0;
3    if (wait_event(&getByte, &tempByte));          // waits for first byte(ack)
4    if (tempByte > 128) {                          // check for ack package
5      if (! ackHandler((tempByte & 0x02) » 1));    // missing error handling
6    }
7    if (tempByte & 0x40) {                         // check for data pack
8      seqBit = tempByte & 0x01;
9      if (wait_event(&getByte, &tempByte));        // waits for second byte(length)
10     dataLength = (int)tempByte;
11     for(i = 0; i < dataLength; i++){             // receives data
12       if (wait_event(&getByte, &tempByte));
13       receiveData[i] = tempByte;
14     }
15     if (wait_event(&getByte, &tempByte));        // wait for third byte(checksum)
16     checksum = 0;
17     checksum = tempByte;                         // get first half of checksum
18     if (wait_event(&getByte, &tempByte));        // wait for fourth byte(checksum)
19     for(i = 0; i < 8; i++){
20       checksum «= 1;
21       if (tempByte & 0x80) checksum |= 0x01;     // get last half of checksum
22       tempByte «= 1;
23     }
24     if (checksumHandler(&receiveData[0], dataLength)) fullPacketHandler();
25   }
26 }
```

Når en hel pakke modtages, beregnes checksummen på ny og sammenlignes med den modtagne. Er de to checksummer ens, sendes en **ACK** pakke tilbage til senderen. Til sidst sættes et flag `readyFlag`, som fortæller brugerfunktionen `getPacket`, at der er data klar i bufferen. For videre udredelse af brugen af denne, se 4.5.

Afsendelse af en datalinkpakke

Datalink laget opbygger en datalinkpakke, som skal sendes til det logiske lag, funktionen `send2logisk()` håndterer dette. Hvis det logiske lag ikke er færdig med at sende foregående data, vil dets buffer(`writeBuffer[]`) være fyldt, og laget vil ikke være i stand til at modtage datapakken fra datalink laget.

Linie 3 i nedenstående kode forsøger at skrive til det logiske lags buffer(`writeBuffer`), hvis dette ikke lykkes ventes 10 millisekunder(linie 4), hvorefter `send2logisk()` kalder sig selv for at forsøge igen. Linie 2 sørger for at dette kun sker tre gange. Det vil sige, at `send2logisk()` får tre forsøg til at sende datapakken videre til det logiske lag. Hvis dette ikke lykkes returnerer den 0(linie 14), en fejlmeddelse som propagerer ud til brugerprogrammet.

```
1  int send2logisk (int lengthOfData){
2    if (retryCount ++ > 3) return 0;
3    if (! write2Buffer (&sendPacket[0], lengthOfData)) {    // sends data to logisk
4      delay(10);
5      send2logisk (lengthOfData);                           // try again
6    }
7    else ackflag = 0;                                       // reset ackflag
8    ackfieldBackup = sendPacket[0];                         // save ackfield
9    ackTimestamp = sys_time;                                // set acktimer
10   if (! ackflag && ackTimer()){                           // start acktimer
11     sendPacket[ ACKFIELD] = 0x00;                         // reset ackfield
12     return 1;
13   }
14   else return 0;
15 }
```

Endvidere kontrollerer `send2logisk()` også, at **ACK** pakker modtages indenfor en tidsperiode på 300 millisekunder(linie 7-12). Vi havde ikke tid til at teste denne tidsperiode for effektivitet, dog har vi testet at den fungerer i praksis.

4.2 Synkronisering af sender og modtager

Før den egentlige overførsel af data kan begynde, er det vigtigt at sender og modtager er synkroniserede på en måde så modtageren ved at den skal modtage en **frame**. Endvidere skal modtageren også vide, hvornår **frame**'en er slut. Til at implementere dette bruger vi en kombination af et start- og slutflag og timing. Dette afsnit fokuserer på vores arbejde med problemstillinger omkring dette.

4.2.1 Implementering af en pausefunktion

Den nedenstående `for`-løkke brugte vi som en pausefunktion. Til at finde ud af, hvor lang en pause, der var lig med en given værdi af $wait$, kørte vi en test, hvor $wait = 10^7$.

```
1  asm("
2    orc #0x80, ccr              ; disable interrupts
3  ");
4
5  for(i = 0;  i < wait;  i++);
6
7  asm("
8    andc #0x7F, ccr             ; enable interrupts
9  ");
```

Vi tog tid på, hvor lang tid RCX'en var om de 10^7 iterationer, gentog forsøget fem gange, og fik følgende resultater:

```
Test 1: 33,886 sek.
Test 2: 33,627 sek.
Test 3: 33,835 sek.
Test 4: 33,992 sek.
Test 5: 33,623 sek.
-----------------------
Gennemsnit: 33,793 sek.
```

Af denne gennemsnitsværdi følger at antallet af iterationer pr. tid er:

$$10^7 it./33,793 sek. = 295919,273 it./sek. \approx 296 it./msek.$$

Dette gav os en pålidelig pausefunktion, som vi let kunne definere pausens varighed på. For en pause på for eksempel 3 sekunder, skal $wait = 296 it/msek * 1000 * 3 sek = 888000 it.$

4.2.2 Implementering af sendersiden

Et vigtigt aspekt i vores logiske lag, er muligheden for at slå **interrupts** fra, når der sendes data. Dette gøres for at undgå at, **scheduleren** skifter sendeprocessen ud, og dermed ødelægger timingen mellem sender og modtager. Til at slå **interrupts** fra eller til, bruges bit 7 i **CCR** registret. Linierne 5 til 7 i kodestumpen slår **interrupts** fra.

Den nedenstående kodestump er taget fra `logisksend_init()`. Linie 4 er det sted, hvor senderens logiske lag venter på, at `writeBuffer[]` er fyldt op, og

den skal gå igang med at sende. Linierne 8- 11 bruges til at fortælle modtageren, at der er data på vej, og til at synkronisere sender og modtager.

Det første der sker, er at senderen sender høj i cirka 54 millisekunder[1], for at fortælle modtageren, at der er data på vej. Herefter sendes lav i perioden `INITWAIT`, som har værdien 1000, hvilket svarer til cirka 3 millisekunder. Efter denne synkronisering, vil en eventuel modtager være klar til at modtage data. Med eventuel mener vi at senderen på nuværende tidspunkt ikke ved om der er en modtager på linien.

While løkken der starter på linie 12, sørger for at sende indtil `writeBuffer[]` er tom. Den efterfølgende **if-else** konstruktion sørger for at kontrollere om den næste bit der skal sendes er en 1-bit eller en 0-bit. Linie 17 og 23 sender henholdsvis en 1-bit og en 0-bit ved hjælp af device driveren. Se 4.1.1

```
1  ...
2  while(1){
3    last = -1;
4    if (wait_event(&startsend,0));                  // wait for data in buffer
5    asm("
6      orc #0x80, ccr                                ; disable interrupts
7    ");
8    send_1();
9    for(i = 0; i < 16000; i++);                     // send start signal (high)
10   send_0();
11   for(i = 0; i < INITWAIT; i++);                  // synchronize
12   while(bufferContents-- > 0) {                   // write till buffer empty
13     if (writeBuffer[bufferPointer++]) {
14       if (last == 1) sendWait = SHORTWAIT;        // decide sync period
15       else sendWait = LONGWAIT;
16       last = 1;
17       send_1();                                   // send high
18       }
19       else {
20         if (last == 0) sendWait = SHORTWAIT;      // decide sync period
21         else sendWait = LONGWAIT;
22         last = 0;
23         send_0();                                   // send low
24       }
25       if (bufferPointer >= BUFFERSIZE) bufferContents = 0; // bufferoverflow
26       for(i = 0; i < sendWait; i++);                 // sync wait
27     }
28   send_0();                                       // reset output port
29   asm("
30     andc #0x7F, ccr                               ; enable interrupts
31   );
32 }
```

[1] Se ovenstående afsnit 4.2.1 for et eksempel på hvordan dette udregnes

4.2.3 Implementering af modtagersiden

A/D konverteren er bindeled mellem det logiske lag og inputporten, og en af det logiske lags opgaver, er at kommunikere med denne. **A/D konverteren** styres af **ADCSR** registret på adresse 0xFFE8. I linie 18 sættes Port 1 til at køre i **single-mode** fordi det i denne mode, er lettere at styre porten. Med dette mener vi, at vi i **single-mode** kan sige helt præcist, hvornår **A/D konverteren** skal foretage en konvertering af det analoge signal på inputporten. Den foretager også kun én konvertering i modsætning til **scan-mode**, som løbende skanner portene.

Tråden standser i funktionens linie 13, og processen går i **waiting** tilstanden. Når `calcSens_new()` returnerer 1, betydende at høj er blevet målt på porten, returnerer `wait_event()` og tråden kan fortsætte, når afvikleren giver processen CPU tid. I linie 13 venter modtageren på, at der læses lav på porten; et signal om at selve dataoverførslen starter, efter en pause med længden `INITWAIT`. Modtagerens `INITWAIT` er en smule større end senderens, for at tage højde for den tid det tager for modtageren at "opdage" lav signalet fra senderen.

Linie 16 er en del af implementationen af fejlhåndteringen. Tælleren `breakCount` holder styr på, hvor mange 0-bit i træk der modtages. Hvis der modtages mere end tre, kan det kun skyldes een af to ting; enten er det fordi **framen** er slut, eller også er der sket en eller anden form for fejl. I henhold til vores design, er slutningen af en **frame** markeret med bit mønstret 111000. Hvis senderen af ukendte årsager blev afbrudt midt i sin transmission af data, ville modtageren læse lav på porten, indtil `breakCount` bliver lig med tre. Når dette sker, afgør `check4packet()` om det var slutningen af en **frame**, eller om der er sket en fejl. Dette sker ved at kontrollere, om der blev modtaget tre gange høj før de tre gange lav. Hvis den finder ud af at der er sket en fejl, droppes pakken(linie 32).

```
1  ...
2    while(1){
3      *adcsr &= 0xD8;                    // stop a/d conversion
4      *adcsr |= 0x01;                    // set scan on port 2-3
5      *adcsr |= 0x20;                    // start a/d conversion
6      breakCount = 0;
7      if (wait_event(&calcSens_new,0));  // wait for startsignal (high)
8      bufferReadPointer = 0;             // reset buffer
9      bufferContents = 0;                // reset buffer
10     asm("
11       orc #0x80, ccr                   ; disable interrupts
12     ");
13     while(calcSens_new());             // wait for sync(low)
14     for(i = 0; i < INITWAIT; i++) ;    // synchronize
15     last = -1;
16     while(breakCount < 3) {            // read until end of frame
17       for(i = 0; i < RECWAIT; i++) ;   // sync wait
18       *adcsr = 0x2A;                   // start a/d conversion
19       for(i = 0; i < 2; i++);          // required pause
20       while(*adcsr < 0x80);            // wait for conversion end
```

```
21          new = (*datareg < 128);                          // read conversion result
22          if ( new == last ) {                             // ensures 2 writes pr bit
23            readBuffer[bufferReadPointer +bufferContents ] = new;
24            bufferContents ++;
25            if (new == 0) breakCount++;                    // checks for endflag/error
26            else breakCount = 0;                           // checks for endflag/error
27            last = -1;
28            if (bufferContents >= BUFFERSIZE) breakCount = 3; //bufferoverflow
29          }
30          else last = new;
31        }
32        if (! check4packet()) bufferContents = 0;          // drop false packet
33        asm("
34          andc #0x7F, ccr                                  ; enable interrupts
35        ");
36        msleep(1);                                         // wakeup
37      }
38  }
```

4.2.4 Tests af INITWAIT - SHORTWAIT - LONGWAIT

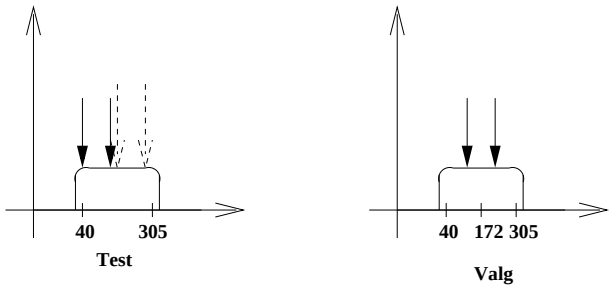
Et af de vigtigste aspekter i en netværksprotokol, er timingen mellem sender og modtager. For at sikre denne timing, er det nødvendigt først at synkronisere dem. I ovenstående kodestump, gøres dette på linie 14. Når sender og modtager er synkroniserede, kan senderen begynde at sende data. Mens den sender data, er der også brug for at opretholde en timing imellem dem. dette gøres på linie 17.

For at finde de optimale venteperioder, har vi gennemført en række tests. Testene er foregået ved, at vi sender 8 bit til modtageren, som derefter udskriver de modtagne bits. Derefter har vi justeret vente værdierne, indtil vi fandt maks og minimums værdierne for en korrekt modtaget pakke.

4.2.5 INITWAIT

For at finde en passende synkroniserings værdi til `INITWAIT`, sendte vi bitstrømmen: 1-0-0-0-0-0-0-0-1. Dermed kunne vi både se, om venteperioden var for lang eller for kort.

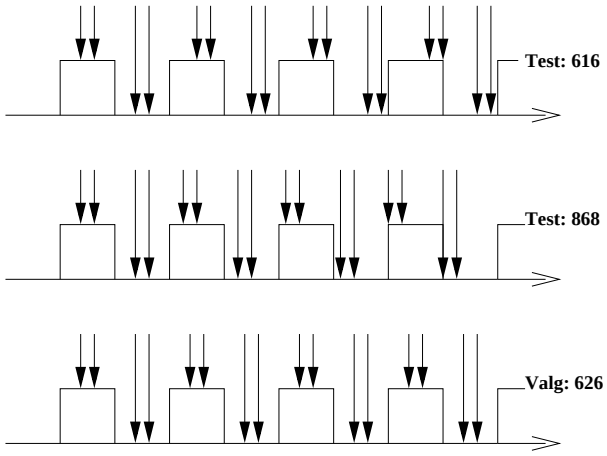
Som det ses på figur 4.1 fandt vi grænseværdierne til 40 og 305. Et gennemsnit af de to tal, giver en passende værdi 172 til `INITWAIT`.



Figur 4.1: Test af INITWAIT

4.2.6 SHORTWAIT og LONGWAIT

Det viste sig, den tid det tager for senderen at sende to høj, ikke er den samme som det tager den, at sende en høj og en lav. Derfor måtte vi definere to seperate timing perioder: `SHORTWAIT` og `LONGWAIT`. Denne timing er vigtigere end synkroniserings timingen med `INITWAIT`. Hvis bare den er lidt for kort eller for lang, vil timingen mellem sender og modtager, forværres for hver sendt bit. Derfor var vi meget omhyggelige med testene omkring disse.



Figur 4.2: Test af LONGWAIT

Figur 4.2 viser, hvordan vores tests faldt ud. En `LONGWAIT` på 616 er en absolut minimums værdi for 8 bit data, og 868 en absolut maksimum. Vi tog gennemsnittet af disse to tal, da det må være så tæt på det optimale som muligt.

Endvidere har vi testet det med 29 byte data, og det fungerede uden fejl i samtlige tests. Altså må vi konkludere, at timingen og synkroniseringen mellem senderen og modtageren er stabil.

4.3 Integrering i LegOS

Vi ønskede først at initialisere vores netværksprotokol sammen med kernen, på samme vis som LNP. Det gav os dog en del problemer. Vi forsøgte at finde den rette placering i LegOS' opstartsface, så protokollen bliver initialiseret, efter at LegOS har startet sine vigtigste dele (såsom scheduleren og **memory manageren**).

Desværre lykkedes det os ikke at finde frem til en løsning. Derfor valgte vi, at initialisere protokollen sammen med et eventuelt brugerprogram. Det kan også sagtens forsvares, eftersom vores protokol er myntet på kommunikation mellem brugerprogrammer. Netværket startes umiddelbart før brugerprogrammet startes. Dette gør vi i LegOS filen *program.c*. Vi har koblet initialiseringen sammen med **run** knappens interrupthandler i funktionen `key_handler()`. Før vi kalder `network_init()`, sørger vi for at resette protokollen. Det er nødvendigt, hvis man vil kunne starte brugerprogrammet igen, efter man har stoppet det.

Når vi starter de tre processor, som vores protokol kører i, tildeler vi dem alle næsthøjest prioritet. Dette er valgt ud fra, at flere dele af protokollen bygger på `wait_event()` funktionen. Så for at denne retunerer hurtigst muligt, er det vigtigt at afvikleren ofte giver processortid til den. Grunden til at vi ikke giver den højest prioritet er, at vi gerne vil kunne stoppe protokollen når brugerprogrammet stoppes. Og handler'en til **run** knappen dræber netop alle processer med næsthøjest prioritet og nedefter.

Hvis ikke protokollen stoppes, når brugerprogrammet stoppes, vil det ikke være muligt at downloade et nyt brugerprogram til RCX'en, uden at slukke den helt med **on-off** knappen.

4.4 Organisering i filer

Vi har delt både det logiske lag og datalink laget op i to filer: en fil med senderdelen og en med modtagerdelen. Dette har vi gjort for at lette overskueligheden. De fire c-filer vi compiler med i kernen er:
logiskesend.c
logiskrec.c
datasend.c
datarec.c

Endvidere har vi også to header filer, som gør at filerne, eller "lagene", kan kommunikere med hinanden:
logisk.h datalink.h

Og endelig er der en header fil, som et eventuelt brugerprogram skal inkludere for at bruge protokollen:
appinterface.h

4.5 Bruger manual

Vores netværks protokol er lavet med henblik på, at give brugeren en alternativ tilgang til inter-RCX kommunikation, i forhold til LNP protokollen. Dette er tænkt som en manual til brugerprogramsudviklere.

4.5.1 Filhåndtering

Brugeren har på nuværende tidspunkt kun brug for at inkludere een fil: *appinterface.h*. Med denne headerfil, har brugeren adgang til de to funktioner, som skal bruges for at sende data vha. vores protokol.

4.5.2 Funktionshåndtering

Som sagt er der to funktioner som brugeren kan tilgå:

int fillSendBuffer(unsigned char *, unsigned char) bruges til at sende data til den anden RCX. Første argument til funktionen, er en pointer til et dataarray. Dette dataarray må kun have elementer af størrelsen een byte. Endvidere er der en grænse på maksimalt 25 byte data. Dvs at det andet argument, som er antallet af elementer i arrayet, ikke må være større end 25. Har man mere data, må man sende det over flere omgange.

Realistisk set vil man nok sjældent ønske, at sende mere end 25 byte ad gangen, mellem to brugerprogrammer. Funktionen returnerer 1 hvis dataene blev sendt og modtaget af den anden RCX. Den returnerer 0, hvis dataene ikke blev både sendt og modtaget.

int getPacket(unsigned char *) bruges til at modtage data med. Funktionen tager som argument en pointer til et array. Dette array skal helst have 25 elementer, da det er den maksimale mængde data der kan overføres ad gangen. Det indkommende data vil være 8-bit data. Funktionen returnerer 0, hvis der ikke er modtaget

noget data, ellers fyldes arrayet, som argumentet peger på, med data, og funktionen returnerer antallet af dataelementer i arrayet.

Det er op til brugerprogrammet at kalde funktionen med jævne mellemrum, da protokollen ikke garanterer, at modtaget data gemmes, indtil brugerprogrammet har hentet det. Hvis en ny mængde data modtages, vil bufferen blive overskrevet.

4.5.3 Mangler

Meget sent i udviklingsfacen (dagen inden aflevering) opdagede vi et problem med sensorportene. Når vores protokol kører på RCX'en, kan man ikke bruge LegOS's drivere til sensorportene. Det er selvfølgelig en uheldig sideeffekt af vores ekstensive manipulering af A/D konverteren. Men vi er opmærksomme på problemet, og vil snarest levere en ny driver.

5 Konklusion

Vi har igennem projektperioden arbejdet med at designe og implementere en netværksprotokol på en RCX klods med styresystemet LegOS. Designet skulle være inspireret af OSI-modellen([Tan96]).

Protokollen skulle implementeres i programmeringssproget C med inline assembler. En del af det forudgående analysearbejde, var at undersøge og forstå, hvordan et indlejret system, som RCX'en, fungerede. Envidere skulle vi opnå forståelse for, hvordan netværksprotokoller designes og implementeres. Dette var specielt med henblik på at forstå, hvordan LNP var implementeret på RCX'en.

Efter at have kigget nøje på sammenhængen i LNP's lag, bemærkede vi, at de services som lagene tilbyder, ikke var særligt skarpt opdelt. Derfor blev et af kravene til vores egen protokol, at den skulle have veldefinerede interfaces. Endvidere ville vi gerne bruge input/output porte og kabler i stedet for IR-kommunikation. Motivationen for dette var, at vi ikke kunne finde dokumentation for, at det var gjort før. Vi mente, at det kunne være interessant at beskæftige os med noget, der ikke var andre, der har prøvet.

Men en ting er analyse og design - noget andet er at realisere en sådan protokolstak. Da vi skulle påbegynde implementationsfasen, stødte vi ind i en række problemer.

I vores systemdefinition (side 4) skriver vi, at vi gerne ville kunne adressere hver enkelt RCX i vores netværk. Dette har vi valgt ikke at implementere i vores protokolstak, da vi primært har fokuseret på det logiske lag og datalink laget. Oprindeligt ønskede vi at designe og implementere flere lag i vores protokolstak, men på grund af problemer med RCX-klodsen og ikke mindst LegOS, fik vi desværre ikke tid til dette. Havde vi haft tid til at videreudvikle på vores stak, ville adressering af enkelte RCX'er i et netværk, få høj prioritet.

Vi ønskede at bruge en enkelt inputport til både input og output, men som beskrevet, viste det sig ikke at være umiddelbart muligt. Desværre brugte vi mange kræfter, og meget tid, i forsøget på at realisere dette. Da vi fik revurderet designet, og blev enige om en fremgangsmåde med at bruge to porte på hver RCX, kom vi ud over denne problemstilling. Denne beslutning burde vi have taget noget før, da en for stor del af projektperioden gik med at forsøge den første løsning. Dette gav udslag i at vi var under tidspres i resten af perioden.

På trods af tidspres, fik vi udviklet og implementeret en fungerende protokol stak. Vi kan garantere, at sendte pakker når frem, og at de er fejlfri. Endvidere leverer vi et let tilgængeligt interface til en evt. bruger.

Gennem analysen af LNP, og ved konstruktionen af vores egen protokolstak, har vi opnået en dybere forståelse for, hvordan en netværksprotokol fungerer. Endvidere har vi også fået detaljeret indblik i, hvorledes styresystemer fungerer. Omkring dette er det vores erfaring, at det at bruge et allerede implementeret styresystem også giver nogle problemer.

Hvis vi på nuværende tidspunkt stod overfor valget om vi ville videreudvikle på et eksisterende OS eller implementere vores eget fra bunden, ville konklusionen nok blive at vi ville implementere vores eget. Denne konklusion er vi kommet frem til efter at have brugt mange ressourcer på at forstå, hvordan forskellige specifikke detaljer af LegOS var implementeret.

Vi har også gjort os nogle erfaringer i, hvordan styresystemer til indlejrede systemer designes. Endvidere har arbejdet med RCX'en også givet os et indblik i hvordan computere virker på det nederste niveau, og hvordan man kommunikerer med dette niveau.

Generelt for os alle er, at dette møde med styresystemer, design af netværk, registre og assembler kode, samt programmeringssproget C, har været vores første. Vi kan derfor sige, at vi har gjort os erfaringer på områder, som før projektperioden var ukendte for os.

Litteratur

[Cap01] Ole Caprani. Rcx manual. Technical report, University of Aarhus, Department of Computer Science, http://www.daimi.au.dk/dArkOS/Vaerktoejer.dir/RCX.vejledning.dir/Vejledning.html, 2001.

[Chr02] Olaf Christ. Tcp/ip enabled legos. Technical report, University of applied sciences Hamburg, http://www.informatik.fh-hamburg.de/~christ_o/, 2002.

[Corwna] Hitachi Corp. H8/300l series - programming manual. Technical report, Hitachi, unknown.

[Corwnb] Hitachi Corp. Hitachi h8 series hardware manual. Technical report, Hitachi, unknown.

[Ef01] Gruppe E2-113-f1a. Chaos - an rcx os where chaos is only in the name. Technical report, AAU, 2001.

[gC00] AAU group C2-203. Rcx operating system. Technical report, Aalborg University, http://www.cs.auc.dk/~mrbeen/dat2/index.html, 2000.

[LV02] Albert Huang Luis Villa. *legOS HOWTO*. http://www.cs.brown.edu/courses/cs148/, 2002.

[Nie00] Stig Nielsson. *Introduction to the legOS kernel*. http://legOS.sourceforge.net, 2000.

[Nor02] Finn Nordbjerg. Slides fra spo-forelæsning. Technical report, Aalborg University, http://www.cs.auc.dk/~kenhans/SPO-F02/garbageCollection.ppt, 2002.

[Sta01] William Stallings. *Operating Systems - Internal and Design Principles*. Prentice Hall, Inc., Upper Saddle River, New Jersey 07458, 2001.

[Tan96] Andrew S. Tanenbaum. *Computer Networks - Third Edition*. Prentice Hall, Inc., Upper Saddle River, New Jersey 07458, 1996.

[Vil00] Luis Villa. *LegOS HOWTO*. http://legos.sourceforge.net/HOWTO/t1.html, 2000.

Tabeller

Figurer