Programmering i maskinkode på *AMIGA*

A.Forness & N.A.Holten

Hæfte 10

Indhold

DATASKOLEN
Postboks 62
Nordengen 18
2980 Kokkedal

Telefon 49 18 00 77

Postgiro 7 24 23 44

LIBRARY OG SYSTEM-FUNKTIONER

I dette brev skal vi beskæftige os en del med OPERATIV-SYSTEMET i AMIGAen, og vi begynder dette kapitel med at forklare hvad LIBRARY- og SYSTEM-FUNKTIONER er.

Ordet LIBRARY betyder på dansk: BIBLIOTEK. I AMIGA findes mange forskellige biblioteker. Hvert af dem indeholder en række funktioner som kaldes SYSTEM-FUNKTIONER. For eksempel indeholder GRAFIK-biblioteket alle funktioner som har med grafik at gøre, mens DOS-biblioteket indeholder alle de funktioner, som har med disk-håndtering og AMIGA-DOS at gøre.

Vi forklarer nu, hvordan man benytter sig af biblioteker når man programmerer.

Studer først FIGUR 1 bagest i brevet. Denne figur viser et udkast over hvordan et bibliotek kan se ud. På adresse $11000 ligger bibliotekets BASE. Fra adresse $11000 og opefter ligger selve funktionerne (eller rutinerne/programmerne), mens der "nedenfor" adresse $11000 ligger en række pointere, som peger på de forskellige funktioner, som ligger over bibliotek-BASEN.

Altså: Når du skal hoppe til en funktion (rutine), finder du dens adresse ved at hente en af pointerne frem - du går altså IKKE DIREKTE til den rutine, som du ønsker at benytte.

Og nu spørger du sikkert - HVOFFER IK'? Jo, for når der kommer en ny version af operativsystemet (KICKSTART) er programrutinerne i biblioteket ofte ændrede (og derved forhåbentlig forbedret). Rutinerne er sandsynligvis blevet kortere end de var før. Dette resulterer i at rutinerne (funktionerne) ligger på andre adresser end de gjorde før.

Så hvis du er gået direkte til rutinerne i dine programmer, vil de ikke virke under nye operativsystem-versioner. Altså: Når COMMODOREs programmører forandrer i et bibliotek, forandrer de også pointerne, mens selve listen med pointere altid ligger på samme sted (i forhold til bibliotek-BASEN).

Vi går et skridt videre og begynder med at vise, hvordan man finder bibliotek-BASEN (adressen) på det såkaldte EXEC-LIBRARY -og det gøres enkelt således:

```
MOVE.L  $4,A6
```

Vi har nu fået adressen på EXEC-BASEN i adresseregistret A6. AMIGA lægger altid adressen på EXEC-BASEN på denne adresse - $04, når maskinen tændes. Vi ved nu, at på adresserne under (nedenfor) den adresse, som ligger i A6, findes pointerne. Vi skal ikke beskæftige os med mere end en funktion fra dette bibliotek lige nu. Denne funktion kaldes "OpenLibrary", og pointeren ligger i OFFSET 408 under/nedenfor BASEN. Et hop til denne funktion vil derfor se således ud:

```
JSR         -408(A6)
```

Denne funktion åbner et bibliotek. Vi har endnu ikke givet besked om, hvilket bibliotek som skal åbnes. Her kommer et program som åbner DOS-biblioteket:

```
MOVE.L     $4,A6
LEA.L      dosname,A1
JSR        -408(A6)
RTS

dosname:
DC.B       "dos.library",0
```

Den første linie lægger EXEC-BASEN ind i A6.

Den anden linie henter adressen på "dosname" ind i A1. Dette register peger nu på en tekst, som vi har deklareret i bunden af programmet (ikke medtaget her). Læg mærke til at vi lægger et nul ind efter sidste bogstav i navnet. Dette skal gøres for at funktionen (OpenLibrary) skal kunne vide, hvor slutningen på teksten befinder sig.

Den tredie programlinie hopper til funktionen "OpenLibrary".

Når rutinen ("OpenLibrary"-funktionen) er udført, ligger BASE-adressen på DOS-LIBRARY i D0.

Altså: "OpenLibrary"-funktionen vil have en pointer til bibliotek-navnet i A1, og returnerer BASE-adressen på dette bibliotek til D0. Hvis ikke "OpenLibrary"-funktionen finder det ønskede bibliotek, returneres 0 i D0. På denne måde kan du sammenligne D0 med 0 for at checke om åbningen blev gennemført eller om der opstod en fejl.

Læg mærke til at alle system-funktioner som returnerer en værdi, lægger denne i register D0. Læg også mærke til, at BASE-adressen, for det bibliotek du skal arbejde med, altid skal ligge i A6. For at klargøre A6 til at arbejde med DOS-biblioteket, kan du lægge denne instruktion efter JSR-instruktionen i programmet ovenfor:

```
MOVE.L     D0,A6
```

De "bibliotek-hop" du gør efter denne instruktion vil nu ske i DOS-biblioteket.

Vi har nu givet dig en enkel indføring i hvordan bibliotekerne (LIBRARY - flertal: LIBRARIES) er opbygget i AMIGA. Vi vil i næste kapitel fortsætte med to andre funktioner fra EXEC-biblioteket.

HUKOMMELSES-ALLOKERING

I dette kapitel skal vi lære om HUKOMMELSES-ALLOKERING (hvordan man sætter hukommelse til side til specielle formål) på AMIGA.

Forestil dig at du har skrevet et program, som opsætter en blank skærm med en størrelse på 320 * 256 PIXELs i et BITPLANE. For at vise denne skærm skal du bruge 10240 BYTEs hukommelse (320 * 256 = 81920 PIXELs, 81920/8 = 10240 BYTEs, 10240/1024 = 10kB). Du skal altså sætte en BUFFER op i programmet på denne størrelse. Ulempen ved at lægge denne buffer (blok) i selve programmet er, at den eksekverbare (kørbare) fil (OBJECT-fil) vil indeholde 10 kB med udelukkende nuller. Det gør filen unødig stor.

En anden metode at gøre det på, er at ALLOKERE hukommelse (sætte hukommelse af) for skærmen. Ordet ALLOKERE er egentlig engelsk (ALLOCATE, og udtales allokeit) og betyder "sætte af".

I og med AMIGAens operativ-system kan håndtere flere programmer ad gangen - såkaldt MULTI TASKING - kan du ikke bruge nogen fast adresse til at lægge en sådan skærm-hukommelse på. Derfor skal vi benytte en system-funktion for at finde ud af, hvor der er ledig plads i hukommelsen. Altså: Du opgiver hvor mange BYTEs du vil have, og hvilken type hukommelse det skal være (CHIP eller FAST). Derefter hopper du til den system-funktion, som returnerer start-adressen på hukommelses-blokken (det sted i hukommelsen, hvor den første BYTE til din skærm findes).

Hvis der ikke er nok ledig hukommelse i maskinen, returneres værdien 0.

Når du har ALLOKERET en hukommelses-blok, vil dette hukommelses-område være sikret mod at andre programmer får tildelt samme område. Vi må derfor frigive hukommelses-blokken før vi afslutter programmet. Hvis ikke vi gør det, vil der blive mindre og mindre hukommmelse tilbage, hvis vi kører programmet flere gange. Du kan til sidst blive nødt til at udføre en RESET (BOOTe op) for at få frigjort maskinens hukommelse igen.

Lad os se på programeksempel MC1001 (som ligger på kursusdiskette nr. 1 - AMIGA MC DISK 1), hvor vi ALLOKERER en hukommelsesblok på 100000 BYTEs.

Linie 1: Lægger værdien 100000 ind i D0.

Linie 2: Hopper til rutinen "allocchip".

Linie 4: Sammenligner værdien i D0 med værdien 0.

Linie 5: Hvis D0 er lig 0, hoppes der til "nomem". Det vil sige, at hvis der ikke findes 100000 BYTEs fri CHIP-hukommelse, vil programmet hoppe til "nomem".

Linie 7: Henter adressen på "buffer" ind i register A0.

Linie 8: Lægger værdien, som ligger i D0 ind på adressen som A0 peger på. I programlinie 22 har vi afsat et LONGWORD til at lægge hukommelses-blokkens startadresse ind i.

Linie 12: Lægger værdien 100000 ind i D0.

Linie 13: Lægger adressen på "buffer" ind i A0.

Linie 14: Henter den værdi som ligger i "buffer" ud og lægger den i A0. Registret A0 vil nu indeholde start-adressen på hukommelses-blokken.

Linie 15: Hopper til "freemem".

Linie 16: Afslutter programmet.

Linie 18
-19: Dette udføres hvis der ikke er nok hukommelse ledigt.

Linie 21
-22: Her ligger det LONGWORD, som start-adressen på hukommelses-blokken lægges i.

Linie:
25-32,
34-40,
42-48: Disse rutiner er helt ens, bortset fra at de ALLOKERER forskellige typer af hukommelse. Den første rutine (allocdef) vil ALLOKERE FAST-hukommelse hvis det findes, hvis ikke ALLOKERER den CHIP-hukommelse. Den anden rutine (allochip) vil altid allokere CHIP-hukommelse, mens den tredie kun vil ALLOKERE FAST-hukommelse.

Vi forklarer kun den første af disse tre rutiner, da de er næsten helt ens.

Linie 26: Lægger alle registre undtagen D0 på STACKen. D0 lagres ikke, fordi der skal returneres en værdi til dette register.

Linie 27
-28: Lægger værdien 1 ind i D1 og bytter om på WORDene i D1. Registret D1 vil nu indeholde $10000.

Linie 29: Lægger værdien som ligger på adresse $000004 ind i A6, som nu vil indeholde BASE-adressen på EXEC-biblioteket.

Linie 30: Hopper til system-funktionen "AllocMem". Funktionen "AllocMem" tager dels imod størrelsen på hukommelses- blokken via register D0 og dels hvilken type hukommelse som skal ALLOKERES via register D1. Værdien $10000 i D1 vil ALLOKERE FAST-hukommelse, hvis det findes - hvis ikke ALLOKERES CHIP-hukommelse. Værdien $10002 vil føre til en ALLOKERING af CHIP-hukommelse, mens $10004 vil ALLOKERE FAST-hukommelse.

Som tidligere nævnt returnerer (eller lægger) denne funktion hukommelses-blokkens startadresse i D0. Hvis der ikke findes tilstrækkelig hukommelse, returneres 0 til registret.

Linie 31: Henter de gamle register-værdier som ligger på STACKen.

Linie 32: Rutinen afsluttes.

Linie 50: Her begynder den rutine, som frigør en hukommelses-blok efter den er blevet ALLOKERET.

Linie 51: Lagrer alle registrene på STACKen. Læg mærke til at denne rutine ikke returnerer nogen værdi.

Linie 52: Lægger værdien som ligger i A0 ind i A1.

Linie 53: Lægger EXEC-BASEN ind i A6.

Linie 54: Hopper til system-funktionen "FreeMem". Denne funktion vil frigøre den hukommelse, som tidligere er blevet ALLOKERET.

Linie 55: De gamle register-værdier hentes ud fra STACKen.

Linie 56: Afslutter rutinen.

Vi har nu set på hvordan man ALLOKERER hukommelse på AMIGA. Dette program (MC1001) benytter ikke den hukommelse som ALLOKERES til noget. Det frigør den bare igen og afslutter kørslen. Det er meningen, at du skal bruge disse rutiner i dine egne programmer.

LÆSNING OG SKRIVNING AF FILER

I dette kapitel vil vi se på, hvordan man læser og skriver til en fil på diskette.

Det er meget nyttigt at kunne håndtere filer når man programmerer. System-funktionerne til dette ligger i DOS-biblioteket, og hedder: OPEN, CLOSE, READ og WRITE. Før man kan komme til enten at læse en fil eller skrive i den, skal filen åbnes. Dette gøres med OPEN-funktionen. Derefter kan du bruge READ- og WRITE-funktionen for at læse eller skrive data til filen. Når du er færdig med at behandle filen, skal den lukkes. Til dette bruges CLOSE-funktionen.

Lad os gå i gang med programeksempel MC1002. Det læser en fil - som hedder "Testfil" - ind i "buffer".

Linie 1: Lægger værdien 24 ind i register D0. Denne værdi angiver den maksimale længde (antal BYTEs) som du vil indlæse. I dette tilfælde ved vi, at filen er 24 BYTEs lang; men du kan sætte denne værdi højere, hvis du vil (husk at udvide størrelsen på bufferen).

Altså: Denne værdi sættes til det højeste antal BYTEs som du vil indlæse. Hvis filen er kortere end den angivne værdi vil hele filen blive indlæst. Hvis filen er længere end den angivne værdi, vil kun det angivne antal BYTEs blive indlæst. Så pas på at du får hele filen med - hvis det er det du ønsker.

Linie 2: Lægger adressen på "filename" ind i A0.

Linie 3: Lægger adressen på "buffer" ind i A1.

Linie 5: Hopper til rutinen "readfile".

Linie 7: Sammenligner værdien i D0 med værdien 0.

Linie 8: Hvis D0 er lig 0, hoppes til "error".

Linie 10: Afslutter programmet.

Linie 12
-13: Der vil blive hoppet hertil, hvis der opstod en fejl under indlæsningen af filen.

Linie 15

-16: Her deklareres navnet på den fil, som skal indlæses. Læg mærke til at navnet skal afsluttes med et 0 (nul).

Linie 18
-19: Her ligger 50 BYTEs, hvori fil-dataene lagres.

Linie 22: Her begynder selve rutinen, som indlæser filen.

Linie 23: Lagrer alle registrene - undtagen D0 - på STACKen. D0 lagres ikke, fordi registret returnerer en værdi.

Linie 24: Lægger værdien som ligger i A0 ind i A4.

Linie 25: Lægger værdien i A1 ind i A5.

Linie 26: Lægger værdien i D0 ind i D5.

Linie 27: Lægger EXEC-BASEN ind i A6.

Linie 28: Lægger adressen på "r_dosname" ind i A1. Registret A1 peger nu på teksten "dos.library".

Linie 29: Hopper til funktionen "OpenLibrary".

Linie 30: Lægger værdien som ligger i D0 ind i A6. Dette register indeholder nu BASE-adressen på DOS-biblioteket (DOS-BASEN).

Linie 31: Lægger værdien 1005 ind i D2. Denne værdi betyder, at der skal læses fra filen.

Linie 32: Lægger værdien som ligger i A4 ind i D1. Registret indeholder nu pointeren til filnavnet.

Linie 33: Hopper til funktionen "Open".

Linie 34: Sammenligner værdien i D0 med 0.

Linie 35: Hvis D0 er 0, hoppes der til "r_error". Altså: Hvis filen ikke findes hoppes der til "r_error".

Linie 36: Lægger værdien som ligger i D0 ind i D1.

Linie 37: Lægger værdien i D0 ind i D7.

Linie 38: Lægger værdien som er i A5 ind i D2.

Linie 39: Lægger D5 ind i D3.

Linie 40: Hopper til funktionen "Read".

Linie 41: Lægger værdien som ligger i D7 ind i D1.

Linie 42: Lægger værdien i D0 ind i D7.

Linie 43: Hopper til funktionen "Close".

Linie 44: Lægger værdien i D7 ind i D0.

Linie 45: Henter de gamle register-værdier, som ligger på STACKen.

Linie 46: Afslutter rutinen.

Linie 47-
50: Hertil hoppes der, hvis filen ikke findes.

Linie 52-
53: Her ligger biblioteknavnet "dos.library" deklareret.

Altså: Rutinen "readfile" skal have følgende værdier ind:

D0 = Maksimal læselængde

A0 = Pointer på filnavnet.

A1 = Pointer til bufferen hvori der skal indlæses data.

Når rutinen "readfile" er udført, indeholder D0 den længde som blev indlæst.

For at køre dette program skal du være i DIRECTORY "BREV10" på kursusdisketten. Det kan kan gøres fra K-SEKA på følgende måde:

Vi antager, at du har lagt kursusdisketten i "df1:".

```
SEKA>vdf1:brev10
```

Du vil nu få en liste over alle de filer, som ligger i "BREV10" DIRECTORY. Blandt disse filer skal der være en fil, som hedder "Testfil". Assembler så programmet, og start det med kommandoen "j".

For at se om filen virkelig er indlæst, kan du gøre følgende:

```
SEKA>qbuffer
```

Du vil nu se en udlistning af hukommelsen der hvor "buffer" ligger.

Var det i orden? Det håber vi, og går igang med forklaringen til programeksempel MC1003.Det er næsten magen til MC1002, så vi omtaler kun de ting, som er forskellige.

Meningen med dette programeksempel er at lave en fil, som hedder "Testfil" og så skrive teksten "Hallo, dette er en test!" ind i den.

Definitionen på rutinen "writefile" ser sådan ud:

IND: D0 = Længden på det som skal skrives.

A0 = Pointer til filnavnet

A1 = Pointer til dataene som skal skrives.

UD: D0 = Længden som blev skrevet.

Som du ser returneres længden, som blev skrevet til D0. Du kan sammenligne D0 med 0 for at finde ud af om der var fejl i skrivningen. Hvis D0 er lig med den værdi, som D0 havde før "writefile"-rutinen blev udført, er alt gået som ønsket.

Vi tror ikke der behøves en mere dybtgående forklaring af dette programeksempel. Det vigtigste er ikke at forstå fuldt ud hvordan rutinerne (allocchip, writefile, osv) er programmeret, men hvordan du bruger dem!

ARGUMENT-INDLÆSNING

I dette kapitel skal vi se nærmere på de såkaldte ARGUMENTER.

Lad os først tage et eksempel. Du har sikkert brugt kommandoen "list" nogen gange. Denne kommando ligger på alle WORKBENCH-disketter i DIRECTORY "c". Denne kommando er egentlig et helt almindeligt program. Når du skriver "list df0:", så er "list" det program som bliver udført, mens "df0:" er en tillægsoplysning, som programmet behøver for at kunne udføre sit job - og det er dette som kaldes et ARGUMENT.

Hvis vi skulle lave et program, som indlæste en fil fra disketten og så skrev den ud på skærmen, ville det være fint at kunne opgive filnavnet direkte på denne måde i CLI:

```
1>myprog testfil
```

Denne metode kan vi også bruge i vores programmer. Hemmeligheden bag det hele er egentlig ganske enkel. Når du starter dit program (OBJECT-filen) fra CLI, vil registret A0 (før programmet startes) indeholde det sted i hukommelsen, ARGUMENTET ligger, mens registret D0 vil indeholde en værdi som fortæller hvor mange tegn det består af (ARGUMENTS-længde).

Lad os tage et kig på - og samtidig forklare - programeksempel MC1004.

Linie 1: Sammenligner D0 med værdien 1. Læg mærke til at det er længden på ARGUMENTET+1 (plus **en**), som lægges i register D0. Altså: Hvis ARGUMENTET er 4 tegn langt, vil D0 indeholde 5.

Linie 2: Hvis D0 er lig med eller mindre end en, hoppes der til "noarg". Altså: Hvis ikke der opgives ARGUMENT, hoppes der til "noarg".

Linie 4: Lægger adressen på "argbuffer" ind i A1.

Linie 5: Lægger værdien som ligger i D0 ind i D7.

Linie 8: Lægger værdien som ligger på den adresse som A0 peger på, ind i den adresse som A1 peger på. Derefter adderes 1 til begge registre (A0 og A1).

Linie 9: Trækker værdien 1 fra register D0. (Man kan også sige det således: Mindsker D0 med 1).

Linie 10: Sammenligner værdien i D0 med 0.

Linie 11: Hvis D0 ikke er lig 0, hoppes der til "copyarg". Altså: Programlinierne 8 til 11 kopierer ARGUMENT-teksten ind i "arg-buffer".

Linie 13-16: Disse programlinier skriver ARGUMENT-teksten ud på skærmen (i cli-vinduet). Vi kommer tilbage senere i dette brev med en forklaring på, hvordan man kan skrive tekst i CLI-vinduet.

Linie 18: Afslutter programmet.

Linie 20-21: Hvis der ikke er opgivet noget ARGUMENT, hoppes der hertil.

Linie 23-24: Her ligger ARGUMENT-bufferen deklareret.

Linie 26-54: Disse programlinier har også med skrivning til CLI-vinduet at gøre, og vil som nævnt før, blive forklaret senere i brevet.

Dette program skal køres udenfor K-SEKA for at det skal give nogen mening (du ser jo ikke meget af CLI-skærmen i K-SEKA). Programeksemplet ligger færdigt som OBJECT-fil på kursusdiskette, og startes således:

```
1>MC1004 Hurra!
Hurra!
1>
```

Du har nu lært hvordan du kan få fat i et ARGUMENT, der kan indtastes samtidig med kommandoen der starter et program. I næste kapitel skal vi kombinere denne ARGUMENT-funktion med indlæsning af en fil i et SCROLL-tekst program.

SCROLL-PROGRAM MED TEKSTFIL

I dette kapitel skal vi se nærmere på programeksempel MC1005, som benytter nogle af de systemfunktioner, vi har behandlet indtil nu.

Programeksempel MC1005 er egentlig en udvidet udgave af programeksempel MC0701 (SCROLL-tekst). Udvidelsen består i at man indlæser teksten som skal SCROLLes fra en fil. Det fungerer på den måde at du bruger fil-navnet som et ARGUMENT. Programmet sørger for, at der ALLOKERES en buffer, som filen derefter bliver indlæst i.

Og så er det selve forklaringen af programeksemplet. Læg mærke til at vi kun forklarer de programlinier, som er nye i forhold til MC0701.

Linie 1: Sammenligner værdien i D0 med værdien 1.

Linie 2: Hvis D0 er større end 1, hopper programmet til "argok". Altså: Hvis der er et ARGUMENT, hop til "argok".

Linie 4: Hvis der ikke findes noget ARGUMENT, afsluttes programmet.

Linie 7: Lægger adressen på "filename" ind i A1.

Linie 9: Her begynder loopen som kopierer ARGUMENTET ind i "filename-bufferen".

Linie 10: Lægger værdien, som ligger på den adresse A0 peger på, ind på den adresse som A1 peger på. Derefter lægges der 1 til (1 adderes til) både A0 og A1.

Linie 11: Trækker 1 fra den værdi som findes i D0.

Linie 12: Sammenligner værdien i D0 med værdien 1. Vi sammenligner D0 med 1, fordi der i slutningen af argumentet ligger en ASCII-kode, som angiver "linieskift" eller "return". Dette tegn skal ikke kopieres ind i "filename"-bufferen og derfor sammenligner vi med 1 for at hoppe over det sidste tegn.

Linie 13: Hvis D1 ikke er 1, hopper programmet tilbage til "copyargloop".

Linie 15: Lægger værdien 50000 ind i D0. (Hvorfor? Se forklaringen til næste programlinie).

Linie 16: Hopper til rutinen "allocdef" som vil prøve at ALLOKERE 50000 BYTEs.

Linie 18: Sammenligner værdien i D0 med værdien 0.

Linie 19: Hvis D0 er forskellig fra 0, gik ALLOKERINGEN godt, og der hoppes til "memok".

Linie 21: Hvis D0 indeholdt 0, afsluttes programmet (Hvilket betyder, at der ikke var tilstrækkelig fri hukommelse).

Linie 24: Lægger adressen på "buffer" ind i A1.

Linie 25: Lægger værdien som ligger i D0 ind i "buffer". Det deklarerede LONGWORD "buffer" vil nu indeholde start-adressen på den hukommelses-blok som blev ALLOKERET.

Linie 27: Lægger adressen på "filename" ind i A0.

Linie 28: Lægger værdien som ligger i D0 ind i A1.

Linie 29: Lægger værdien 50000 ind i D0. (Se forklaringen til næste programlinie).

Linie 31: Hopper til rutinen "readfile". Denne rutine vil nu indlæse filen "filename" i den ALLOKEREDE buffer med maksimal længde på 50000.

Linie 33: Sammenligner værdien i D0 med værdien 0.

Linie 34: Hvis D0 er lig 0, hoppes der til "freeup". Altså: Hvis der skete en fejl ved indlæsningen af filen, hoppes der til "freeup". Vi skal hoppe til "freeup" for at frigøre hukommelsen, som vi ALLOKEREDE, før programmet afsluttes.

Linie 36-77: Disse linier er identiske med de tilsvarende i programeksempel MC0701, og behøver derfor ikke forklares her.

Linie 79: Her frigøres den ALLOKEREDE hukommelse.

Linie 80: Lægger værdien 50000 ind i D0.

Linie 81: Lægger adressen på "buffer" ind i A0.

Linie 82: Værdien som ligger i "buffer" lægges ind i A0. Altså: Her lægges adressen på den ALLOKEREDE hukommelses-blok ind i A0.

Linie 83: Hopper til rutinen "freemem".

Linie 85-162: Disse programliner er allerede forklaret i programeksempel MC0701.

Linie 164
195: Denne rutine er den samme rutine, som er forklaret i et tidligere kapitel i dette brev. Det er denne rutine, som indlæser en fil.

Linie 197
212: Disse to rutiner ALLOKERER og frigører hukommelse - og er forklaret i et tidligere kapitel i dette brev.

Linie 214
330: Disse linier går igen i MC0701, og behøver ingen forklaring her (håber vi...).

Linie 332
333: Her ligger det LONGWORD, som indeholder startadressen på den hukommelses-blok, som ALLOKERES.

Linie 335
336: Her har vi deklareret 50 BYTEs, som skal indeholde fil-navnet.

For at køre dette program kan du bruge den færdige objekt-fil, som ligger på kursusdisketten. BOOT derfor kursusdisketten op og skriv følgende:

```
1>MC1005 BREV10/Text
```

Vi har, som du ser, lavet en lille fil, som ligger i DIRECTORY "BREV10". Hvis du vil prøve at lave din egen tekst, så kan du bruge en enkel tekst-editor som f.eks "ED". Husk at lægge et "*"-tegn (en stjerne) i slutningen af teksten, sådan at SCROLL-programmet kan finde ud a,f hvor teksten slutter. Lykke til!

I næste kapitel skal vi beskæftige os med noget der er kompliceret og spændende (men ikke vanskeligere); nemlig hvordan man læser og skriver direkte på "grund-niveau" til en diskette.

LÆSNING OG SKRIVNING DIREKTE TIL DISK

I dette kapitel gennemgår vi læsning fra og skrivning til diskette, direkte på SECTOR-niveau.

Vi starter med at forklare, hvordan data er lagret på en diskette.

Studer FIGUR 2 bagest i brevet. En AMIGA-diskette er opdelt i 80 spor - som ringe i vandet, såkaldte TRACKs. Begge sider på disketten bruges til at lagre data på. Derfor kan vi sige at en diskette har 160 spor. Disse spor er inddelt i smådele - SECTORer. På hvert TRACK findes 11 SECTORer. Ialt findes derfor 11 * 160 SECTORER på en AMIGA-diskette. Hver SECTOR indeholder 512 BYTEs. Den totale lagringskapacitet bliver da 11 * 160 * 512 = 901120 BYTEs (det vil sige 880kB).

For at finde det sted på disketten, hvor vi skal læse/skrive, skal vi altså bruge: spor, side og SECTOR. For at gøre dette enklere, bruger AMIGAen betegnelsen BLOCK (kaldes også DISK-BLOCK således at man ikke forveksler det med en hukommelses-blok - MEMORY BLOCK) til at angive en position på disketten. Lad os tage nogle eksempler:

SPOR	SIDE	SECTOR		BLOCK
0	0	0	=	0
0	0	1	=	1
0	0	2	=	2
0	0	10	=	10
0	1	0	=	11
0	1	10	=	21
1	0	0	=	22
1	0	1	=	23
10	0	0	=	220
40	0	0	=	880
40	1	0	=	891
70	0	0	=	1540
79	0	0	=	1738
79	1	0	=	1749
79	1	10	=	1759

Som du ser her, har en AMIGA-diskette ialt 1760 BLOCKe (nummereret fra 0 til 1759). På BLOCK 0 og 1 ligger den såkaldte BOOT-BLOCK (egentlig BOOT-BLOCKe). Når du tænder for maskinen og sætter en diskette i, blive disse to blokke automatisk indlæst i hukommelsen (mere om dette i BREV XII). På BLOCK 880 (spor 40) og et stykke udover (kommer an på hvor mange filer som findes på disketten) ligger DIRECTORIET. Ordet DIRECTORY kan oversættes med "kartotek" eller "indholdsfortegnelse". Det er her AMIGA leder for at finde ud af hvilke filer, som findes og hvor dataene i en fil ligger på disketten.

Lad os nu se lidt på programeksempel MC1006: Vi vil ikke bryde vores hjerner med at forklare selve rutinen, som læser/skriver til disketten. Det er meget vigtigere at lære at bruge rutinen.

Her kommer registrene som benyttes:

IND: **A0 = Pointer til læse- eller skrive-buffer.**
D0 = Disk-station.
D1 = Start-BLOCK.
D2 = Længde (antal BLOCKe).
D3 = Mode.

Mode: **1 = READ**
2 = WRITE (til diskbuffer)
3 = UPDATE

Som du ser returnerer rutinen ingenting. A0 skal altså indeholde adressen på bufferen, som der skal læses eller skrives fra.

Registret D0 angiver hvilken diskette-station som skal benyttes. Du angiver et tal fra 0 til 3, som tilhører de respektive stationer:

"DF0:", "DF1:", "DF2:" og "DF3:".

D1 angiver start-BLOCKen, og D2 angiver længden (i BLOCKs), som skal læses eller skrives.

Registret D3 angiver om du skal læse eller skrive. Værdien 1 angiver læsning, mens værdien 2 angiver skrivning.

Den sidste værdi (3) er lidt speciel. Lad os først fortælle at AMIGA benytter en såkaldt DISK-buffer. Forestil dig, at du har indlæst BLOCK 0 og så ændrer du lidt på dataene. Derefter skriver du den tilbage på disketten (ved at benytte mode 2). Du vil så opdage, at diskette-stationen ikke reagerer. Dette sker, fordi BLOCK 0 ligger i DISK-bufferen (i hukommelsen). Når du skriver BLOCKen tilbage, vil AMIGAen lægge dataene i DISK-bufferen, og ikke direkte ud på disketten. Derfor skal du først bruge mode 2 til at lægge de nye data i DISK-bufferen. Derefter benytter du mode 3 til at opdatere disketten med de nye data, som nu ligger i DISK-bufferen.

Vi fortsætter med en test af dette programeksempel. Det første du skal gøre er at FORMATERE en ny diskette. **Hvis du bruger en diskette med filer på, kan disse filer gå tabt.** Brug derfor en ny diskette.

Når du er færdig med at formatere den diskette du vil bruge, starter du K-SEKA og henter programeksemplet ind.

Derefter forandrer du programmet således:

```
lea.l     buffer,a0
move.l    #0,d0
move.l    #100,d1
move.l    #1,d2
move.l    #2,d3


bsr       sector

move.l    #3,d3

bsr       sector
rts

....

buffer:
dc.b      "Dette er en test"
blk.b     512,0
```

Når du har ændret dette i programmet, kan du assemble det. Sæt derefter den FORMATTEREDE diskette i "DF0:" og kør programmet. Hvis alt gik godt skulle disklampen lyse ca. 1 sekund.

Vi har altså skrevet disse data ud på disketten. For at indlæse BLOCKen igen forandrer du programmet således:

```
lea.l     buffer,a0
move.l    #0,d0
move.l    #100,d1
move.l    #1,d2
move.l    #1,d3

bsr       sector
rts

....

buffer:
blk.b     512,0
```

Assemble programmet og kør. Hvis du vil se de indlæste data på skærmen skriver du følgende:

```
SEKA>qbuffer
```

Du lagde sikkert mærke til at diskettelampen ikke lyste. Det er fordi BLOCKen som vi lige netop havde skrevet, ligger i disk-bufferen. Derfor behøver AMIGAen ikke diskettestationen for at få fat i dataene. Prøv at tage disketten ud og sætte den ind igen. Derefter kører du programmet en gang til (assembler først). Nu vil du se, at diskettestationen starter op.

Vi har nu set på hvordan du kan læse og skrive direkte til DISK-BLOCKene (SECTORerne). I næste kapitel skal vi se lidt nærmere på, hvordan man kan skrive til og læse fra CLI-vinduet.

LÆSNING OG SKRIVNING TIL CLI

I dette kapitel skal vi gennemgå INPUT og OUTPUT til CLI-vinduet.

Læsning og skrivning (INPUT og OUTPUT) til CLI-vinduet udføres næsten på samme måde som ved læsning og skrivning af filer. I stedet for at åbne en fil bruger vi to andre funktioner, som kaldes INPUT og OUTPUT. Disse gør samme nytte som OPEN, men de åbner egentlig ikke noget som helst. Disse funktioner finder det allerede åbnede CLI-vindue frem. Hvis vi benyttede OPEN-funktionen, ville AMIGA åbne et nyt vindue på skærmen. Dette er jo ikke meningen når man skriver til CLI-vinduet.

Vi har som sædvanlig lavet komplette rutiner for både at læse og skrive til CLI-vinduet. Her kommer opsætningen for disse to rutiner.

readchar:

```
IND:    A0 = Pointer til en buffer.
        D0 = Maksimal læselængde.

UD:     D0 = Aktuel længde.
```

writechar:

```
IND:    A0 = Pointer til en buffer.
        D0 = Længde.

UD:     D0 = Aktuel længde.
```

For rutinen "readchar" (læs tegn) skal A0 indeholde adressen på den buffer, som tastatur-dataene skal indlæses i. Register D0 angiver den maksimale mængde data du ønsker skal kunne indlæses, mens registret returnerer den faktiske længde, som blev indlæst. En indlæsning afsluttes ved et tryk på "RETURN" eller ved at det maksimale antal tegn er indtastet. Læg mærke til at tegnene som indlæses også bliver vist på skærmen (CLI-vinduet).

I den anden rutine som hedder "writechar" (skriv tegn) skal A0 indeholde adressen på teksten, som skal skrives ud. Registret D0 skal indeholde længden på teksten, og returnerer den længde (det tegnantal) som blev skrevet.

Vi har en rutine i programmet som vi ikke har omtalt endnu. Den hedder "opendos". Det eneste denne rutine gør, er at åbne "dos.library" og så lagre DOS-BASEN i LONGWORDet, som er deklareret i slutningen af programmet ("txt_dosbase"). Denne rutine skal kun udføres en gang i programmet og skal udføres

før du kan benytte rutinerne "readchar" og "writechar".

Lad os nu se lidt nærmere på programeksempel MC1007. Vi forklarer ikke instruktionerne, men holder os til brugen af dem.

Linie 1: Hopper til "opendos".

Linie 3: Lægger 40 ind i D0. Dette angiver den maksimale længde, som kan indlæses fra tastaturet.

Linie 4: Lægger adressen på bufferen "input" ind i A0.

Linie 5: Hopper til rutinen "readchar".

Linie 7: Adderer værdien 7 til værdien, som findes i register D0. Hvis du studerer de to buffere "output" og "input" på programlinie 12-16, vil du se at teksten "Hallo," indeholder 7 tegn. Når vi nu lægger 7 til i D0, vil registret indeholde længden på "hallo"-beskeden - plus længden på teksten som blev indlæst.

Linie 8: Lægger adressen på "output" ind i A0.

Linie 9: Hopper til rutinen "writechar". Dette vil resultere i at beskeden "Hallo, ?????????" vil komme ud på skærmen.

Linie 10: Afslutter programmet.

Linie 12
13: Her ligger teksten "Hallo," deklareret.

Linie 15
16: Her har vi deklareret 40 BYTEs til at lagre den tekst, som bliver indlæst fra tastaturet.

Linie 17: Den er ny! Denne kommando er meget enkel. Den sikrer, at instruktionerne, som ligger efter denne havner på en lige adresse. Hvis du prøver at lægge en MC68000 instruktion på en ulige adresse (f.eks. adresse $10001), vil assembleren give en fejlmelding. Årsagen til dette er at MC68000 har en 16 BITs adresse-BUS (to BYTEs).

Linie 20-
32: Her ligger rutinen "readchar".

Linie 34-
46: Her ligger rutinen "writechar".

Linie 48-
62: Her ligger rutinen "opendos".

Nu er vi klar til at prøve dette programeksempel. At køre dette program fra K-SEKA vil ikke give noget resultat. BOOT derfor kursusdisketten op (DISK 1) og skriv:

```
1>MC1007
```

Tast dit navn ind, og tryk på RETURN. Du vil nu se at AMIGAen "siger hallo" til dig.

Du har nu lært, hvordan du kan læse og skrive til CLI-vinduet. Husk, at det vigtigste er at bruge rutinerne fremfor at forstå dem instruktion for instruktion.

MASKINKODE IX

I dette maskinkodekapitel skal vi gennemgå to instruktioner som hedder **EXG** og **CMPM**.

Vi starter med den enkleste først, nemlig **EXG** (som står for **EX**CHANG**E**, og betyder: "bytte om"). Denne instruktion bytter om på værdierne i to registre. Læg mærke til at den altid bytter om på alle 32 BIT. Den kan bytte både dataregister til dataregister, dataregister til adresseregister, adresseregister til adresseregister og adresseregister til dataregister.

Her kommer nogle eksempler:

```
EXG       D0,D1

EXG       D0,A6

EXG       A0,A2
```

Den bytter altså om på indholdet i registrene. Denne instruktion skulle være let nok at forstå.

Lad os gå over til den anden instruktion **CMPM**, som er en variant af CMP-instruktionen. **CMPM** er en forkortelse for **COMP**ARE **M**EMORY.

Vi demonstrerer det med et lille programeksempel:

```
move.l    #$10000,a0
move.l    #$20000,a1

loop:
cmpm.w    (a0)+,(a1)+
beq.s     loop
rts
```

Dette programeksempel vil sammenligne værdierne, som ligger på henholdsvis adresse $10000 og $20000. Hvis disse værdier er ens, vil den hoppe op igen og sammenligne værdierne på adresse $10002 og $20002, osv. Når den finder to forskellige værdier afsluttes programmet.

Du kan teste forskellige tal ved at bytte BEQ-instruktionen ud med en BNE-instruktion. Prøv selv - glem ikke at det er meget vigtigt at du prøver dig frem. Du lærer meget af dine egne fejl.

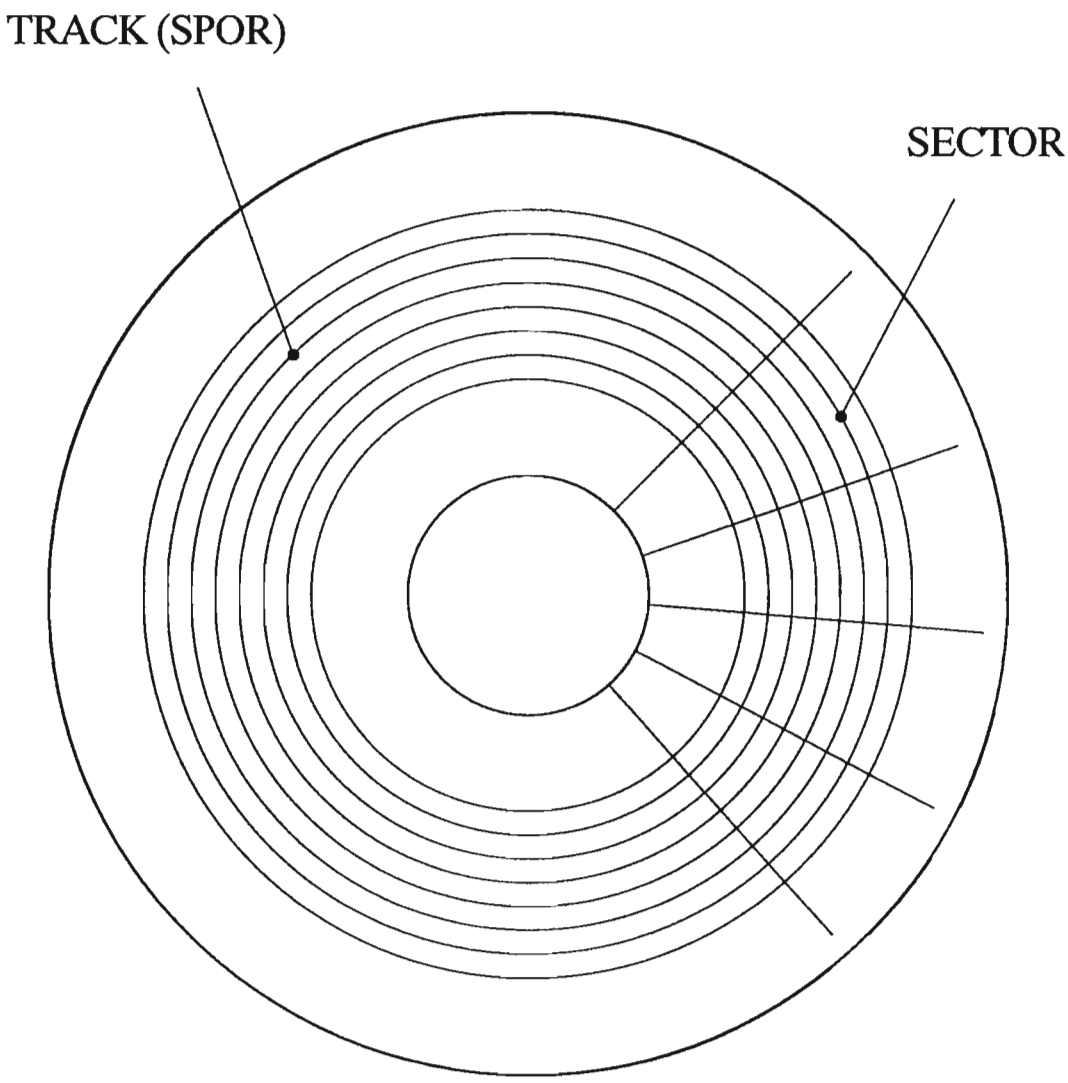
LØSNINGER TIL OPGAVERNE I BREV IX

Opgave 0901: Fordelen ved at benytte INTERRUPTs er, at processoren (MC68000) slipper for at bruge tid på f.eks at checke om du trykker på tastatur-et.

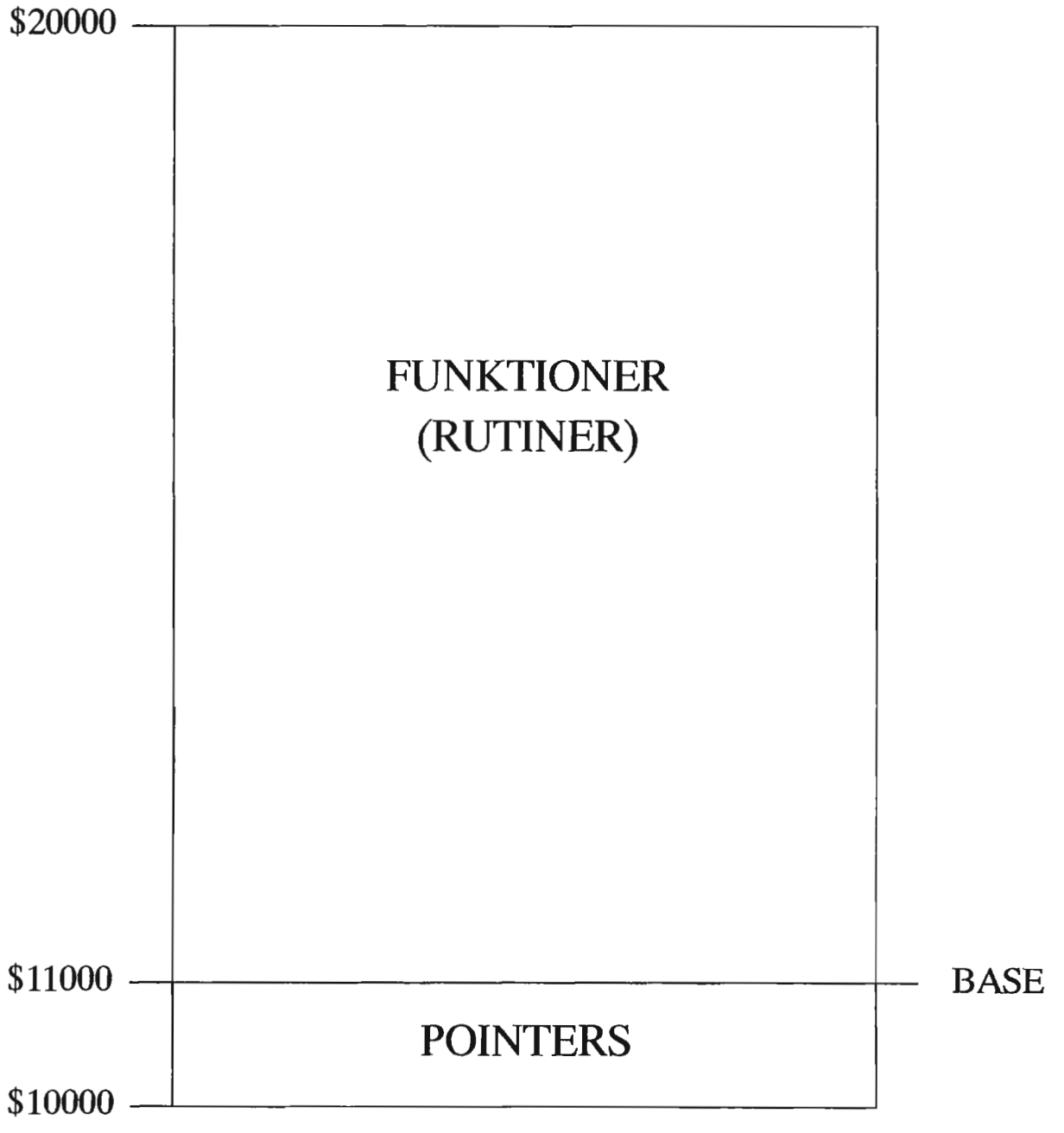
Opgave 0902: INTERRUPT 7 kaldes ofte for NONMASKABLE INTERRUPT.

Opgave 0903: BCHG-instruktionen inverterer (ændrer fra 0 til 1 eller omvendt) en enkel BIT i et register eller hukommelsen.

Opgave 0904: Registret D1 vil indeholde : $6C9D4651



Figur 2.



Figur 1.