Programmering i maskinkode på *AMIGA*

A.Forness & N.A.Holten

Hæfte 9

Indhold

DATASKOLEN
Postboks 62
Nordengen 18
2980 Kokkedal

Telefon 49 18 00 77

Postgiro 7 24 23 44

INTERRUPT I

Vi begynder dette kapitel med at forklare rent generelt hvad INTERRUPT (oversat: AFBRUD) er. Vi vil benytte betegnelsen INTERRUPT fordi det er den, som bruges mest af programmører.

Hvis muligheden for INTERRUPTs ikke fandtes i datamaskiner, ville det give os et problem, som i overført betydning ville udarte sig sådan:

Du sidder og venter på at telefonen skal ringe, men da ringefunktionen i telefonen ikke virker, kan du ikke høre hvis den ringer. Du skal altså løfte røret hele tiden og sige "hallo?" - og, hvis ingen svarer, så lægge på igen.

Måske sidder du ovenikøbet og læser et eller andet - eller gennemgår dagens post på kontoret. Den manglende ringefunktion vil resultere i, at du ikke får læst mere end nogle få sætninger, før du atter må løfte røret af telefonen for at checke om nogen ringer. Hvis du dertil har flere telefoner, og må løfte rørene af konstant, får du ikke læst ret meget. - ret irriterende, ikke?

Et eksempel på INTERRUPTs i en datamaskine kan f.eks være aflæsningen af tastaturet. Hvis vi ikke kunne benytte os af INTERRUPTs, måtte vi checke tastaturet hele tiden for at være sikker på at datamaskinen fik alle vores anslag med.

Men fordi der er en INTERRUPT-funktion, behøver programmet ikke at checke tastaturet i det hele taget. Indlæsningen bliver udført automatisk af en såkaldt INTERRUPT-RUTINE. Denne rutine vil blive udført så snart der trykkes en tast ned på tastaturet. I dette tilfælde er det almindeligt, rutinen lægger tegnet ind i en KEYBOARD-BUFFER - det vil sige at datamaskinen opsamler tegnene, som du indtaster i et lille område i hukommelsen, som den har sat af til den slags. Hvis for eksempel et program er igang med at hente data fra en diskette, kan det altså afslutte diskoperationen før det henter eventuelle tastetryk, som du har lavet i mellemtiden - og som altså ligger lagret i KEYBOARD-BUFFERen.

Dette burde være en god indledning til begrebet INTERRUPT. I næste kapitel går vi videre med hvordan INTERRUPTs virker på din yndlings-maskine: AMIGA.

INTERRUPT II

I AMIGA'en findes der 7 forskellige INTERRUPTs. De er nummereret fra 1 til 7. Det syvende INTERRUPT kan ikke slås fra (og derved forhindres udført), og kaldes derfor NONMASKABLE INTERRUPT. De øvrige INTERRUPTs kan du slå fra, hvis du vil. Så hvis du vil bede maskinen om at slå et INTERRUPT fra, f.eks det som aflæser tastaturet, så er det muligt.

Hvert INTERRUPT i AMIGA'en har en bestemt PRIORITET (eller "forkørselsret"). Lad os se lidt nærmere på netop dette:

INTERRUPT	FUNKTION I AMIGA
7	Eksternt (udvendigt) INTERRUPT
6	Eksternt INTERRUPT
5	Disk
	Seriel port
4	Lyd
3	Blitter
	Vertikal-blank
	Copper
2	IND/UD porte og TIMERs
1	Benyttes som SOFTWARE-INTERRUPT
	Disk
	Serieport

INTERRUPT 7 kan du se helt bort fra. Dette fordi at der i skrivende stund ikke findes noget udstyr, som benytter sig at dette INTERRUPT. Vi koncentrerer os derfor om INTERRUPTene som er nummereret fra 1 til 6.

INTERRUPTene er inddelt i PRIORITETer, hvor INTERRUPT 7 har højeste PRIORITET, og INTERRUPT 1 har laveste PRIORITET. Dette virker således, at imens INTERRUPT 3 udføres, kan et INTERRUPT med lavere eller samme PRIORITET ikke afbryde det som foregår.

Derimod kan INTERRUPTs med høj PRIORITET afbryde et med lavere prioritet for at udføre sin egen rutine først, for derefter at lade den INTERRUPT-RUTINE med lavere PRIORITET (som blev afbrudt) gøre sig færdig. Det er konstrueret sådan, fordi enkelte enheder (disk, BLITTER, COPPER, osv.) kan have brug for lidt hurtigere opmærksomhed end andre enheder.

Læg også mærke til at f.eks INTERRUPT 3 i tabellen har flere funktioner. Dette medfører at INTERRUPT-RUTIENEN skal checke hvilken enhed som genererede INTERRUPTet - på godt dansk: Bad om opmærksomhed - for at få at vide hvad INTERRUPT-RUTINEN skulle udføre. Mere om dette senere.

Lad os nu gå et skridt videre og fortælle om, hvordan forskellige INTERRUPTs "opstår":

INTERRUPT 6: Dette INTERRUPT er, ligesom INTERRUPT 7, også koblet til eksterne enheder. Det betyder at INTERRUPTet ikke bliver brugt, hvis du ikke har harddisk e.l.

INTERRUPT 5 (DISK): Dette INTERRUPT bliver genereret (eller aktiveret) når et såkaldt DISK SYNC WORD er fundet på disketten. Vi kan forklare dette i al enkelhed: Dataene på en diskette bliver lagret i ringe på disketten - på såkaldte spor. For at AMIGAen skal vide, hvor begyndelsen på ringen befinder sig, er der lagret en speciel kombination af enere og nuller (16 BITs) på dette sted. Når AMIGAen finder denne kombination på diskette, vil den hoppe til INTERRUPT-rutinen, som starter DISK-DMA'en for at begynde indlæsningen af data. Dette skal gå hurtigt, og derfor har denne funktion fået en høj PRIORITET.

INTERUPT 5 (Seriel port): Dette INTERRUPT bliver aktiveret når BUFFEREN til seriel-porten er fuld. Serieporten er et INTERFACE (dansk: mellemled), som kan sende eller modtage data fra andre enheder som. f.eks MIDI, et MODEM, en anden datamaskine - eller lignende. Det specielle ved seriel-overføring er, at der bliver overført en BIT ad gangen. Når serieporten har modtaget 8 BITs (en BYTE) skal et BUFFER-register tømmes umiddelbart før næste BIT kommer ind, så det ikke "løber over", - det som kaldes OVERFLOW. Altså: når BUFFER-registret er fuldt, vil dette INTERRUPT genereres (aktiveres) og der hoppes til en INTERRUPT-rutine, som lægger værdien over i BUFFER-registret et sted i RAM, hvor det kan ligge trygt. Som du ser skal også denne funktion have en høj PRIORITET for at være sikker på, at dataene på serieporten bliver modtaget korrekt.

INTERRUPT 4: Dette INTERRUPT genereres når en lydkanal er færdig med at afspille en SAMPLE. Dette betyder, at vi kan lade INTERRUPT-rutinen slukke for AUDIO-DMA'en så SAMPLEn kun bliver spillet en gang, istedet for at lade SAMPLEn blive spillet om og om igen ukontrolleret. Læg mærke til at alle fire lydkanaler er "koblet" på dette INTERRUPT, således at du skal finde ud af hvilken lydkanal, der er kommet til afslutningen af SAMPLEn.

INTERRUPT 3 (BLITTER): Dette INTERRUPT bliver genereret når en BLITTER-operation er færdig med et job. I vores program-eksempler bruges disse (nu velkendte) linier for at checke dette:

```
wait:
BTST      #6,$DFF002
BNE       wait
```

Den anden metode er at lave en INTERRUPT-rutine, som f.eks opsætter og starter næste BLIT selv.

Som du ser har dette INTERRUPT ikke så høj PRIORITET. Dette kommer sig af, at denne operation ikke behøver en så hurtig

opmærksomhed (eller behandling) som f.eks den serielle port i INTERRUPT 5.

INTERRUPT 3 (vertikal-blank): Dette INTERRUPT genereres hver gang elektronstrålen hopper op til linie 0. Altså, dette INTERRUPT udføres hver gang skærmen opdateres. Det har ingen speciel funktion i sig selv, men INTERRUPT-rutinen kan f.eks opdatere positionen til WORKBENCH-pilen og andre ting, som skal udføres for hver skærmopdatering. Vi kunne for eksempel lave en INTERRUPT-rutine, som opdaterer en SCROLL-tekst. Da ville hovedprogrammet se således ud:

```
main:
BTST        #6,BFE001
BNE         main
```

i stedet for.....

```
main:
MOVE.L      $DFF004,D0
ASR.L       #8,D0
ANDI.L      #$1FF,D0
CMP.W       #0,D0
BNE         main

BSR         scroll

BTST        #6,$BFE001
BNE         main
```

INTERRUPT 3 (COPPER): Dette INTERRUPT kan genereres af COPPERen. Du kan for eksempel lade COPPERen vente på linie 200, og derefter generere et INTERRUPT. I praksis benyttes dette INTERRUPT i samme tilfælde som INTERRUPTet ovenfor (vertikal-blank), men med den forskel at du selv kan bestemme ved hvilken skærmposition INTERRUPTet skal startes (altså, ikke fast på linie 0 som vertikal-blank INTERRUPTet er). Du kan også foretage flere INTERRUPTs for hver skærmopdatering, hvis du ønsker det.

INTERRUPT 2: Dette INTERRUPT genereres af en IND/UD-enhed, som styrer parallelporten, mustasten, nogle af diskfunktionerne, osv. Vi kommer tilbage med forklaring af dette INTERRUPT i BREV 11, når vi skal gennemgå denne IND/UD-enhed.

INTERRUPT 1 (SOFTWARE): Dette INTERRUPT er lidt specielt. Det bliver nemlig styret fra selve programmet som køres. Som du sikkert forstår, har dette INTERRUPT ikke nogen speciel funktion, men vi kan nævne en funktion, det bliver benyttet til - nemlig MULTITASKING. Dette betyder - som du måske allerede ved - at to eller flere opgaver (programmer) kan køres samtidig. Dette er ikke helt sandt, fordi det som egentlig sker er, at processoren skifter imellem at køre (udføre) programmerne hele tiden.

Eller: når et program er blevet udført et lille stykke tid (tiden kan variere), så vil programmet selv genere dette INTERRUPT. INTERRUPT-rutinen sørger så for at der hoppes til næste program. Således vil denne INTERRUPT-rutine fordele "processor-kraften" på de forskellige programmer.

INTERRUPT 1 (DISK): Dette INTERRUPT genereres når DISK-DMA'en er færdig med at indlæse data fra en diskette. Denne INTERRUPT-rutine kan for eksempel flytte på læse/skrivehovedet for at kunne starte op på en ny indlæsning. Det har en væsentlig lavere PRIORITET end disk-funktionen i INTERRUPT 5. Det er lavet således fordi denne funktion ikke behøver en så hurtig behandling.

INTERRUPT 1 (SERIEL PORT): Dette INTERRUPT bliver genereret når BUFFERen til den serielle port er tom. Altså: Når maskinen har sendt 8 BITs (læg mærke til at seriel-porten kan sættes op til at sende/modtage 9 BITs ad gangen også), udføres en INTERUPT-rutine, som kan indlægge næste BYTE, som skal sendes ud på serieporten.

Vi har nu gennemgået funktionerne for de enkelte INTERRUPTs i AMIGA. I det følgende kapitel skal vi se på, hvordan man blandt andet laver sin egen INTERRUPT-rutine.

INTERRUPT III

Vi starter dette kapitel med en tabel over indholdet i hukommelses-adresserne $000000 - $0003FF.

ADRESSE	BESKRIVELSE
$000000	Reset, SSP
$000004	Reset, PC
$000008	BUS ERROR
$00000C	ADRESS ERROR
$000010	ILLEGAL INSTRUCTION
$000014	DIVISION BY ZERO
$000018	CHK INSTRUCTION
$00001C	TRAPV INSTRUCTION
$000020	PRIVILEGE VIOLATION
$000024	TRACE EXCEPTION
$000028	UNIMPLEMENTED INSTRUCTION (1010)
$00002C	UNIMPLEMENTED INSTRUCTION (1111)
$000030	ikke benyttet

```
...

$000060                 SPURIOUS INTTERUPT
$000064                 INTERRUPT 1 AUTO-VECTOR
$000068                 INTERRUPT 2 AUTO-VECTOR
$00006C                 INTERRUPT 3 AUTO-VECTOR
$000070                 INTERRUPT 4 AUTO-VECTOR
$000074                 INTERRUPT 5 AUTO-VECTOR
$000078                 INTERRUPT 6 AUTO-VECTOR
$00007C                 INTERRUPT 7 AUTO-VECTOR

$000080                 TRAP#0
$000084                 TRAP#1
$000088                 TRAP#2
$00008C                 TRAP#3
$000090                 TRAP#4
$000094                 TRAP#5
$000098                 TRAP#6
$00009C                 TRAP#7
$0000A0                 TRAP#8
$0000A4                 TRAP#9
$0000A8                 TRAP#10
$0000AC                 TRAP#11
$0000B0                 TRAP#12
$0000B4                 TRAP#13
$0000B8                 TRAP#14
$0000BC                 TRAP#15

$0000C0                 Ikke benyttet
...

$000100                 USER INTERRUPT VECTOR

...
$0003FC                 USER INTERRUPT VECTOR
```

Vi koncentrerer os om adresserne $64 til $7C. Disse hukommelses-områder indeholder pointere (VECTORer) til de forskellige programrutiner, som udføres når en INTERRUPT bliver udført. Altså: Når processoren (MC68000) får et signal af typen INTERRUPT 1, vil den hente værdien, som ligger på adresse $64 (LONGWORD), og derefter behandle denne værdi som en adresse, som den skal hoppe til.

Processoren gør også et par andre ting før den hopper til INTERRUPT-rutinen. Det første den gør, er at flytte værdien, som ligger i programtælleren (PC) på STACKen. Derefter lægger den STATUS-registret på STACKen, og til sidst hopper den til INTERRUPT-rutinen. Altså: Den skal lagre programtælleren således at den ved, hvor den fortsætte når INTERRUPT-rutinen er udført (dette foregår på samme måde som med BSR). Derefter skal STATUS-registret (registret som indeholder bl.a ZERO-flaget, CARRY-flaget, osv), lagres.

Vi belyser det nærmere med et eksempel:

```
CMP.W       #25,D0
BEQ         loop
```

Som vi allerede ved, kan et INTERRUPT ske når som helst, helt ufhængig af andre programmer. Forstil dig at et INTERRUPT skete mellem to instruktioner i eksemplet ovenfor. Hvis D0 f.eks var 25 ville denne CMP-instruktion sætte ZERO-flaget til 1. Når INTERRUPTet så bliver udført før BSR-instruktionen, kan ZERO-flaget have fået en anden værdi (på grund af INTERRUPT-rutinen) før processoren hopper tilbage for at gå løs på BEQ-instruktionen. Dette kan medføre at BEQ-instruktionen vil hoppe "forkert" enkelte gange. Derfor er det nødvendigt at processoren også lagrer STATUS-registret.

Alt dette går automatisk, således at du ikke ikke behøver at tænke over dette når du programmerer en INTERRUPT-rutine. Derimod findes der en anden ting du skal tænke på, nemlig dataregistrene og adresseregistrene. Disse bliver nemlig ikke lagret automatisk under et INTERRUPT. Derfor skal det første, du gør i en INTERRUPT-rutine være at lagre de registre, du skal bruge, på STACKen. hvis du f.eks skal bruge dataregistrene D0 til D3 og adresseregistrene A0 til A1 i rutinen, kan en sådan instruktion for at lagre disse registre på STACKen se således ud (se i maskinkodekapitlet, hvor vi forklarer denne variant af MOVE):

```
MOVEM.L   D0-D3/A0-A1,-(A7)
```

Når INTERRUPT-rutinen er gennemført, lægger du til sidst følgende instruktion ind for at få værdierne tilbage i registrene.

```
MOVEM.L   (A7)+,D0-D3/A0-A1
```

Instruktionen som benyttes til at afslutte en rutine er sædvanligvis en RTS. Men hvad gør denne RTS-instruktion egentlig? Jo, den henter en værdi fra STACKen (et LONGWORD), og bruger denne værdi til at hoppe tilbage til, der den kom fra. Men, vi sagde jo, at når et INTERRUPT opstår, lagres ikke kun programtælleren (PC), men også STATUSREGISTRET. Derfor kan vi ikke benytte RTS for at afslutte en INTERRUPT-rutine. Derimod bruger vi en instruktion som hedder RTE. Denne instruktion vil også sørge for STATUS-registret. Derfor vil strukturen på en INTERRUPT-rutine se således ud:

```
interrupt:
MOVEM.L   D0-D7/A0-A6,-/A7)

.....

MOVEM.L   (A7)+,D0-D7/A0-A6
RTE
```

Vi gennemgår nu programeksempel MC0901, som ligger på kursusdisketten DISK 1.

Linie 1: Lægger adressen på "jump" ind i A1. Læg mærke til at LABELen "jump" peger på en instruktion (programlinie 42).

Linie 2: Lægger værdien, som ligger på adresse $68, ind på adressen som A1+2 peger på. Denne MOVE flytter altså en værdi ind i selve instruktionen på programlinie 42: Først henter vi værdien ud, som ligger på adresse $68. Denne adresse indeholder pointeren (VECTORen) til INTERRUPT 2. Denne værdi lægges så ind i JMP-instruktionen på programlinie 42. JMP-instruktionen udfører et hop i lighed med BRA. Forskellen på disse to er, at BRA hopper med en OFFSET fra selve instruktionen, mens JMP hopper direkte til en opgivet adresse (fast adresse, se også maskinkodekapitlet senere i brevet). Som du ser på linie 42, hopper denne JMP-instruktion til adresse $0. Når vi har udført denne MOVE, vil JMP-instruktionen have fået ny adresse - nemlig adressen på den gamle INTERRUPT-rutine.

Linie 4: Lægger værdien 100 i D0.

Linie 6: Lægger værdien 1 i D1.

Linie 7: Bytter om på WORDene i D1. Dette resulterer i at D1 indeholder $10000.

Linie 8: Lægger værdien fra adresse $4 ind i A6.

Linie 9: Dette er en ny instruktion. JSR betyder **J**UMP to **S**UB **R**UTINE, eller hop til under-rutine (se også i maskinkode-kapitlet). I BREV X kommer vi næremere ind på JSR-kommandoer. I al enkelthed udfører programlinierne 4 til 9 en HUKOMMELSES-ALLOKATION på 100 BYTEs. Det vil sige, at vi beder operativsystemet om at sætte 100 BYTEs hukommelse af til os. På den måde fås et sted i hukommelsen, hvor INTERRUPT-rutinen kan ligge sikkert. Tænk ikke så meget over det lige nu - det er vigtigere at forstå, hvordan INTERRUPT-rutinen virker, end at vide, hvor i hukommelsen den lægges.

Linie 11: Lægger værdien fra D0 ind i A1. Den rutine (HUKOMMELSES-ALLOKATION) som blev udført af programlinie 9, returnerer en adresse fra D0, som peger på den første BYTE i den 100 BYTEs store "blok" som blev ALLOKERET (afsat).

Linie 12: Lægger værdien i D0 ind i D7.

Linie 14: Lægger adressen på "interrupt" ind i A0.

Linie 15: Lægger 24 ind i D0. Den bruges som en tæller.

Linie 18: Lægger værdien, som A0 peger på, ind på adressen, som A1 peger på. Derefter adderes værdien 4 til værdierne i både A0 og A1 (HUSK! LONGWORD).

Linie 19: Trækker værdien 1 fra D0 og checker om D0 er -1. Hvis ikke hoppes der tilbage til "copyloop". På den måde kopieres rutinen "interrupt" (programlinie 28 til 42) ind i den "blok" med hukommelse, som vi ALLOKEREDE tidligere.

Linie 21: Slukker alle INTERRUPTs.

Linie 22: Lægger værdien som ligger i D7 ind på adresse $68. Altså: Adressen på hukommelses-blokken, som vi ALLOKEREDE, bliver lagt ind i pointeren til INTERRUPT 2. Læg mærke til at vi på forrige linie slukkede alle INTERRUPTs. Dette skal gøres før man kan lægge en ny værdi ind i INTERRUPT-pointerne.

Linie 23: Tænder alle INTERRUPTs igen. Nu udføres INTERRUPT-rutinen når der kommer et signal til INTERRUPT 2. Når vor INTERRUPT-rutine er udført, vil den ikke returnere til hovedprogrammet - men ved hjælp af JMP-instruktionen på linie 42 - hoppe til den gamle INTERRUPT-rutine (som tilhører operativsystemet). Dette gør vi for at holde operativsystemet intakt. Vi har altså bare "sneget" vores egen INTERRUPT-rutine ind før operativsystemets egen INTERRUPT-rutine. Dette kaldes for at LINKe en rutine ind (som kan oversættes med at "koble sig ind på).

Linie 25: Afslutter programmet (OBS! INTERRUPT-rutinen afsluttes ikke).

Linie 27: Her ligger INTERRUPT-rutinen.

Linie 28: Lagrer værdien i D0 på STACKen. Vi behøver ikke en MOVEM-instruktion her, fordi vi kun behøver at lagre et register. MOVEM bruges kun når vi vil lagre flere registre ad gangen.

Linie 29: Lægger værdien som ligger på adresse $BFEC01 ind i D0. Denne adresse indeholder koderne, som sendes fra tastaturet når der trykkes på en tast. Vi vil komme nærmere ind på dette register i BREV XI.

Linie 30: Inverterer D0.

Linie 31: Roterer D0 BIT-vis en gang til højre. Se forklaring på denne instruktion i maskinkode-kapitlet.

Linie 33: Sammenlign D0 med 59.

Linie 34: Hvis D0 ikke var 59, hop til "wrongkey". Altså: Programlinierne 29 til 34 aflæser tastaturet og checker om en speciel tast er blevet trykket ned.

Linie 36: Denne instruktion betyder **BIT CHANGE**. Den inverterer BIT 1 på adresse $BFE001. BIT 1 på denne adresse styrer "POWER"-lampen som hhv tændt og slukket. Denne instruktion vil altså resultere i at hver gang den bliver udført, vil den tænde eller slukke lampen. Denne instruktion er også forklaret i maskinkode-kapitlet.

Linie 39: Henter det gamle indhold i D0 ud fra STACKen.

Linie 42: Hopper til den gamle INTERRUPT-rutine.

For at køre dette program, skal du som sædvanlig assemble det. Derefter behøver du bare kommandoen "j" for at starte det.

Når du har kørt programmet, kan du prøve at trykke på tasten "F10" og samtidig se på "POWER"-lampen.

Med dette afslutter vi kapitlerne om INTERRUPTs. Vi kan nævne at i BREV XII kommer nogle lidt mere komplicerede INTERRUPT-rutiner (for MIDI) som vil tage sig af automatisk afsendelse og modtagelse på SERIEL-porten.

MASKINKODE VIII

I dette kapitel gennmgår vi følgende instruktioner:

MOVEM, JMP, JSR, BCHG, RTE, ROR og ROL.

Lad os starte med instruktionen MOVEM. Det ekstra bogstav "M" i denne instruktion betyder MULTIPLE. (kan oversættes med "flere"). Altså: Denne MOVE flytter flere ting ad gangen og benyttes ofte til at lagre og hente fra STACKen:

Denne ene programsætning.....

```
MOVEM.L   D0-D2,-(A7)
```

...udfører det samme som disse 3 linier:

```
MOVE.L    D0,-(A7)
MOVE.L    D1,-(A7)
MOVE.L    D2,-(A7)
```

Et eksempel til...

```
MOVEM.L   (A7)+,D1/A0-A4
```

Det første eksempel lagrer dataregistrene D0, D1 og D2 på STACKen. Det andet eksempel - som udfører det samme som det første - tager både mere plads og tager længere tid.

Det tredie eksempel henter D1, A0, A1, A2, A3 og A4 ud fra STACKen. Du kan også bruge andre registre end A7 til at lagre og hente data. På denne måde kan du lave din egen STACK. Et sådant program kan f.eks se således ud:

```
LEA.L     mystack,A0

MOVEM.L   D0-D5/A1-A4,-(A0)

...

MOVEM.L   (A0)+,D0-D5/A1-A4

...

mystack:
BLK.B     100,0
```

Den næste instruktion vi skal se på er JMP. Denne instruktion betyder **JUMP**, og oversættes til: hop. Forskellen på denne instruktion og BRA, er at BRA-instruktion hopper relativt (den hopper i forhold (OFFSET) til, hvor programtælleren peger). JMP-instruktionen fungerer ikke på samme måde: Lad os vise det med et eksempel:

```
JMP       $FC00D2
```

Den hopper altid til en fast adresse. Altså, BRA-instruktionen bruges oftest når der skal hoppes indenfor samme program, mens JMP mest bruges for at hoppe udenfor programmet (til et andet program).

Næste instruktion på listen er JSR. JSR betyder **JUMP TO SUB RUTINE**, og kan oversættes til: hop til under-rutine. Indtil nu har vi kun brugt BSR-instruktionen fordi vi kun har hoppet til under-rutiner, som ligger indenfor vort eget program. Hvis vi vil udføre et hop til en under-rutine, som ligger udenfor programmet bruger vi altså JSR. Udover at kunne hoppe til en fast adresse som JMP-instruktionen kan, kan JSR indirekte hoppe med et adresseregister. Der kan også specificeres en OFFSET i tillæg til dette. Her kommer nogle eksempler:

```
JSR       $5000
```

og et eksempel på et indirekte hop...

```
JSR       (A0)

JSR       20(A0)
```

Det første eksempel udfører et hop til adresse $5000.

I det andet eksempel bruges værdien, som ligger på adressen som A0 peger på som hop-adresse. Altså: Hvis A0 indeholder $10000 - og der på adresse $10000 ligger værdien $50000, vil der blive hoppet til adresse $50000.

I det sidste eksempel vil der blive hoppet til den adresse, som ligger på adresse A0 + 20. Med andre ord; Den henter værdien som ligger på adresse $10000 + 20 = $10014, og bruger denne værdi som hop-adresse.

Næste instruktion hedder BCHG, og betyder **BIT CHANGE**. Denne instruktion bruges til at invertere en speciel BIT på en adresse eller i et register. Her kommer nogle eksempler:

```
D0=%00101101

vi udfører...

BCHG     #3,D0

...og D0 bliver:

D0=%00100101

Vi udfører samme instruktion en gang til...

BCHG     #3,D0

...og D0 bliver igen:

D0=%00101101
```

Denne instruktion burde være indlysende.

Den næste instruktion har vi forklaret en del om i forklaringen til programeksempel MC0901, nemlig RTE. RTE betyder **RETURN FROM EXCEPTION**. Vi skal ikke rode dig ud i for meget med at prøve at forklare, hvad EXCEPTION er i dette brev, men indskrænker os til at forklare, hvordan instruktion fungerer. I lighed med RTS-instruktionen fungerer denne også som afslutning af en rutine. Forskellen er, at vi bruger RTE for at afslutte en INTERRUPT-rutine. Den tekniske forskel er at RTS-instruktionen kun henter den gamle program-tæller ud fra STACKen, mens RTE-instruktionen derudover henter den gamle værdi, som var i STATUS-registret.

De to sidste instruktioner vi skal forklare er ROR og ROL. De betyder henholdsvis **ROTATE RIGHT** og **ROTATE LEFT**, og oversættes med: roter til højre og roter til venstre. Disse instruktioner er meget lig med LSR og LSL. Den eneste forskel er, at den BIT som bliver roteret ud i den ene ende, kommer ind igen i den anden ende af BIT-gruppen.

Lad os atter vise nogle eksempler:

D=%11010010

Vi prøver denne...

```
ROR.B      #3,D0
```

...og D0 bliver nu sådan:

D0=%01011010

Vi tager en til:

D0=%0110100101011011

efter at følgende er udført...

```
ROL.W      #1,D0
```

... bliver D0 sådan:

D0=%1101001010110110

Det kan også nævnes, at når du opgiver antal skift som en konstant (sådan som i eksemplerne ovenfor), er 8 den højeste værdi som er lovlig. Hvis du skal rotere f.eks 13 BITs, kan du enten bruge to instruktioner, eller benytte dig af følgende metode:

```
MOVEQ     #13,D1
ROL.L     D1,D0
```

Vi håber det var til at forstå - ellers prøv en gang til.

LØSNINGER TIL OPGAVER I BREV XIII

Opgave 0801: AMPLITUDE er højden (styrken) på en lydbølge.

Opgave 0802: SAMPLING er i dataverdenen en lydbølge (bølgeform), som repræsenteres (opbevares) DIGITALT.

Opgave 0803: En A/D-CONVERTER er en elektronisk opkobling, som laver et ANALOGT signal om til et DIGITALT (BINÆRT) signal.

Opgave 0804: Resultatet i D1 bliver: 225 ($000000E1)

OPGAVER TIL BREV IX

Opgave 0901: Hvad er fordelen ved at benytte sig af INTERRUPTs?

Opgave 0902: Hvad kaldes INTERRUPT 7 ofte på AMIGA?

Opgave 0903: Hvad udfører BCHG-instruktionen?

Opgave 0904: Hvad vi D1 indeholde efter at disse instruktioner er udført (prøv uden K-SEKA):

```
MOVE.L      #$64E918AB,D1
ROR.L       #3,D1
ROL.W       #1,D1
ROR.B       #5,D1
```

LILLE DATAORDBOG

AMPLITUDE	Amplitude er højden på f.eks en lydbølge.
SAMPLE	En SAMPLE i forbindelse med lyd er lydbølgedata, som er lagret DIGITALT.
A/D-CONVERTER	En A/D-CONVERTER er en elektronisk opkobling som ændrer (konverterer) et ANALOGT signal til et DIGITALT (BINÆRT) signal.
DIGITAL	DIGITALT er en betegnelse for et signal som kan være enten 0 eller 1 (tændt eller slukket).
ANALOG	Et ANALOGT signal er et signal som kan have uendelig mange værdier.
MIDI	MIDI er en forkortelse for MUSICAL INSTRUMENT DIGITAL INTERFACE som kan oversættes til "DIGITALT MELLEMLED MELLEM MUSIKINSTRUMENTER". Dette system bliver brugt til f.eks at lade et instrument styre et andet. MIDI er mest udbredt på tangent-instrumenter (elektriske pianoer, SYNTHESIZERs, orgler osv.)
INTERRUPT	Ordet INTERRUPT betyder: AFBRYD, og er en Funktion, som ligger i HARDWARE. Benyttes til håndtering af I/O-enheder.
I/O	IN/OUT enheder i computeren, der modtager/sender data IND/UD (f.eks en disk)
NONMASKABLE	Dette ord bliver brugt sammen med INTERRUPT. Et NONMASKABLE INTERRUPT er et INTERRUPT, som ikke kan kobles ud (forhindres).
PRIORITET	Ordet PRIORITET kan oversættes med "fortrinsstilling". Dette bliver ofte brugt sammen med INTERRUPTs. Et INTERRUPT med højere PRIORITET anses for at være vigtigere end et med lavere PRIORITET.
VECTOR	VECTOR kan i data-sammenhæng oversættes til "pointer". Det er ofte en adresse på en underrutine eller en data-tabel.

KOMMENTARER TIL BREV IX

Nu når du er færdig med dette brev, har du været igennem 3/4 af kurset i AMIGA MC. VI antager at du er kommet så langt at du nu kan udføre en hel del avancerede ting i ren maskinkode på din yndlings-maskine.

I dette brev har vi gennemgået INTERRUPTs. Efterhånden som du kommer til at bruge denne funktion, når du programmerer, vil du erfare, at det er en meget følsom funktion. Det sker derfor let at det går galt, hvis du ikke passer på (GURU MEDITATION !)

Det gælder altså om at holde operativsystemet i AMIGA intakt. Sørg derfor, for at du har en back-up før du prøver om det virker.

Glem ikke at DEBUGGE (fjerne fejl) dit program mens du programmerer. Det er næsten umuligt at finde fejlene igen - og stedet hvor de opstod - når dit program er blevet så stort, at det dækker flere A4-sider i ulistet form.

Sørg også for at du lader din kildekode (SOURCE CODE) indeholde rigelig med kommentarer, således at du til enhver tid ved hvad de forskellige rutiner udfører. Brug kommentarer som er lette at forstå. Det kan være vanskeligt at huske om korte ord som "sprcol" skal betyde SPRITE-COLLISION (eller var det SPRITE-COLOR?) når du vender tilbage til dette sted i dit program efter at have programmeret andre dele af programmet.

Det er også vigtigt at du holder en jævn programmeringshastighed i din programmering. Med andre ord: SMED MENS JERNET ER VARMT. Det er lettere at skrive overskuelige programmer på denne måde, samtidig som de bliver enklere at DEBUGGE.

I næste BREV - BREV X - kommer vi ind på et helt nyt område, nemlig AMIGA's operativsystem, hukommelses-allokeringer, læsning/skrivning af filer til disk og læsning/skrivning til CLI-vinduet. Det er de (for programmører) vigtigste og mest brugte system-funktioner. I praksis viser det sig, at man sjældent får behov for flere systemfunktioner end dem vi har nævnt her.

Og netop derfor er det vigtigt at kunne. Desuden vil du - efter at du har læst BREV X - lettere forstå hvordan andre system-funktioner virker, når (eller hvis?) du opsøger speciallitteratur i emnet.

Vi ønsker dig fortsat god fornøjelse!