Programmering i maskinkode på *AMIGA*

A.Forness & N.A.Holten

Hæfte 5

Indhold

DATASKOLEN
Postboks 62
Nordengen 18
2980 Kokkedal

Telefon 49 18 00 77

Postgiro 7 24 23 44

SPRITES 1

En SPRITE (udtales: sprajt og betyder spøgelse eller ånd) et objekt dvs et lille billede, som kan vises og flyttes rundt p} skærmen uafhængig af BITMAPen (eller resten af skærmen). AMIGA har 8 SPRITEs, som kan vises på en gang. En SPRITE kan indeholde 4 farver (med lidt trylleri op til 16 farver - mere om det i BREV XII). Bredden er fast 16 PIXELs, men højden definerer man selv. WORKBENCH-pilen er et eksempel på en SPRITE.

Vi går nu i gang med et eksempel. Læg mærke til at program-eksemplet ikke er vist i dette brev. Du skal bruge KURSUS-DISKETTE 1 og K-SEKA, for at kunne følge med. Derfor bør du indlæse programmet MC0501 ved hjælp af K-SEKA og have programlisten på skærmen, mens du læser forklaringen nedenfor:

Programeksempel MC0501:

Linie 1: Denne MOVE slukker for alle INTERRUPTs (afbrydelser). Hele systemet "fryses" således at det kun er vores program, som udføres. INTERRUPTs vil blive gennemgået og forklaret i BREV IX.

Linie 3-4: Hvis man starter et program direkte fra diskette, og det slukker alle INTERRUPTs, vil diskettestationen ikke standse af sig selv når programmet er læst ind. Vi løser dette problem ved at gøre det inde i selve programmet. Dette vil blive forklaret indgående i BREV X.

Linie 6-17: Disse linier sætter en skærm op (BITMAP) og bør være kendt stof.

Linie 19-24: Lægger SPRITEns adresse ind i COPPERLISTen i SPRITE's SPRITEPOINTER (SPRITEne er nummereret fra 0 til 27). SPRITEPOINTERen er ligesom BITPLANE-POINTERen opdelt i HIGH- og LOW-adresser.

Linie 26-39: Henter adressen på den "blanke" (eller utegnede) SPRITE og lægger den i COPPERLISTen på SPRITEPOINTERne til SPRITE 1-7. Man skal opsætte alle 8 SPRITEs, også selv om du kun vil bruge en.

Linie 41-51: Disse linier henter adressen til skærmen og lægger den i COPPERLlSTen. Derefter startes COPPER-DMA'en, BITPLANE-DMA'en og SPRITE-DMA'en. Læg mærke til programlinie 50. En MOVE til denne adresse har du måske ikke set før. Den er ikke helt nødvendig, men den bør være med for at sikre at COPPERLlSTen startes rigtigt på alle AMIGAer (også fremtidige).

Linie 53-58: Denne lille rutine får processoren til at vente med at gå videre indtil elektronstrålen når linie 0. Husk at linie 0 ikke er BITMAPens øverste linie men at det er den øverste linie, elektronstrålen kan nå (øverste linie på BITMAPen er som regel linie 44 ($2C). Forestil dig at linie 0 ligger ovenover monitorens "plastikkant". Skærmlinierne 0-19 kaldes ofte for VERTICAL-BLANKing.

Linie 60-65: Denne rutine venter til elektronstrålen når linie 1. Grunden til at vi først parkerer på linie 0, og derefter på linie 1 er, at hvis vi kun venter på linie 20, risikerer vi at få en ujævn bevægelse på SPRITEn. Hovedrutinen udføres så hurtigt, at når der hoppes tilbage til programlinie 53 (wait:), er der en vis risiko for at elektron-kanonen stadig er optaget af at tegne linie 0. Som du forstår er AMIGA en **hurtig** datamaskine!

Linie 67: Denne instruktion hopper til movesprite-rutinen, og hopper tilbage igen, når processoren møder en RTS. For den som har lidt kendskab til BASIC: denne instruktion kan sammenlignes med GOSUB, og RTS-instruktionen kan sammenlignes med RETURN. Instruktionen vil blive gennemgået i maskinkodekapitlet i dette brev.

Linie 69-70: Checker om venstre musknap er trykket ned. Hvis ikke, hoppes der tilbage til "wait:".

Linie 72: Slukker COPPER-DMA'en.

Linie 74-76: Henter adressen på den gamle COPPERLIST (den som hører til WORKBENCHen) og lægger den i COPPER-POINTERen.

Linie 78: Starter COPPER-DMA'en igen.

Linie 80: Denne MOVE aktiverer alle INTERRUPTs igen.

Linie 81: Afslutter programmet.

Linie 84-94: Denne rutine flytter SPRITEn på skærmen. Se forklaringen nedenfor.

Linie 97-130: COPPERLlSTen.

Linie 132-133: Her er vort skærmbillede defineret.

Linie 136-152: SPRITE-dataene er defineret her. (forklares nedenfor.)

Linie 154-155: Her er den "blanke" SPRITE defineret.

Studer SPRITE-dataene i eksemplet. Som du ser er det første LONGWORD (2 WORDS) sat til 0. Dette LONGWORD indeholder den position, SPRITEn skal have på skærmen. Linierne 84-94 opdaterer disse tal, således at SPRITEn bevæger sig nedover skærmen. Lad os nu se lidt på SPRITE-dataene:

```
LONGWORD 1: Position og højde på SPRITEn
LONGWORD 2: Grafikdata
LONGWORD 3: Grafikdata
LONGWORD 4: Grafikdata
...(så høj SPRITE som du selv ønsker)
LONGWORD ?: Skal være 0 (sidste datalinie)
```

Vi opdeler det første LONGWORD i BYTEs:

```
LONGWORD: $00 00,$00 00
BYTENR.:   0  1   2  3
```

BYTE 0: Indeholder BIT 0-7 for den vertikale (lodrette) position af SPRITEns øverste linie.

BYTE 1: Indeholder BIT 1-8 for den horisontale position af SPRITEns venstre kant.

BYTE 2: Indeholder BIT 0-7 for den vertikale position af SPRITEns nederste linie.

BYTE 3: BIT 0 indeholder BIT 0 for den horisontale position af SPRITEns venstre kant.

BIT 1 indeholder BIT 8 for den vertikale position af SPRITEns nederste linie.

BIT 2 indeholder BIT 8 for den vertikale position af øverste linie på SPRITEn.

BIT 3-6 benyttes ikke.

BIT 7 kommer vi tilbage til i BREV XII. Denne BIT bruges til at angive "16 farvers" SPRITE (1 = 16 farver og 0 = 4 farver).

Som du ser benyttes der 9-BITs-værdier for at sætte positionerne på SPRITEn. Som du allerede ved, kan en BYTE kun indeholde 8 BITs.

Derfor bliver den overflødige BIT lagt i BYTE 3. Læg mærke til, at horisontalpositionen i BYTE 1 indeholder BIT 1 - 8 og ikke 0-7, således at BIT 0 bliver den overflødige BIT, som lægges i BYTE 3.

Eksempel på positioner: $5080,$5500

SPRITEns øverste linie havner på linie $50, eller 80 DECIMALT. Venstre kant havner på $80 * 2 (256 DECIMALT). Vi skal multiplicere positionen med 2 fordi BIT 0 er den overflødige BIT. Højden på SPRITEn bliver $55 - $50 (altså 5 PIXELs eller linier). Som tidligere nævnt er bredden på en SPRITE altid 16 PIXELs. Det bør også her understreges at SPRITEs altid er i LORES (LOW RESOLUTION = lav opløsning) uanset hvilken skærmopløsning, der benyttes.

$5080,$5501

Dette eksempel vil placere SPRITEn på samme sted som i eksemplet ovenfor, men med den undtagelse at den horisontale position vil være 257 ($101).

I det følgende forklares opstillingen (eller udseendet) af SPRITENs PIXEL-data:

Et LONGWORD er en linie (16 PIXELs) med SPRITEdata. Vi opdeler LONGWORDet i to WORDs (som i programeksemplet). Som tidligere nævnt kan en SPRITE vise 4 farver (3 + baggrund). Hvis vi sammenligner med BITPLANEs, vil første WORD være BITPLANE 1, og andet WORD, BITPLANE 2. Her følger nogle eksempler:

$0000,$0000 = En linie (16 PIXELs) med SPRITECOLOR 0.

$FFFF,$0000 = En linie med SPRITECOLOR 1.

$0000,$FFFF = En linie med SPRITECOLOR 2.

$FFFF,$FFFF = En linie med SPRITECOLOR 3.

Farverne i SPRITEne er inddelt således:

SPRITE nr.	SPRITECOLOR	FARVEREGISTER	TYPE
0 & 1	0	SDFF180	COLOR 00
0 & 1	1	SDFF1A2	COLOR 17
0 & 1	2	SDFF1A4	COLOR 18
0 & 1	3	$DFF1A6	COLOR 19
2 & 3	0	$DFF180	COLOR 00
2 & 3	1	$DFF1AA	COLOR 21
2 & 3	2	$DFF1AC	COLOR 22
2 & 3	3	$DFF1AE	COLOR 23
4 & 5	0	$DFF180	COLOR 00
4 & 5	1	$DFF1B2	COLOR 25
4 & 5	2	$DFF1B4	COLOR 26
4 & 5	3	$DFF1B6	COLOR 27
6 & 7	0	$DFF180	COLOR 00
6 & 7	1	$DFF1BA	COLOR 29
6 & 7	2	$DFF1BC	COLOR 30
6 & 7	3	$DFF1BE	COLOR 31

Som du ser deler SPRITEne samme farveregister to og to. Læg også mærke til at farve 0 i alle SPRITEs bliver FARVEREGISTER 0 (Det er baggrundsfarven på skærmen).

Et eksempel vil vise hvordan SPRITE-farverne opfører sig. (Lad os sige at dette gælder SPRITE nr. 5).

SPRITEdata: $FFFF,$0000 = En linie (16 PIXELs) i SPRITE nr. 5 bliver den farve som FARVEREGISTER 25 indeholder ($DFF1B2, COLOR 25). Se også figur 3 bagest i dette brev.

SPRITES II

Vi starter dette kapitel med opsætningen til SPRITE-POINTERs.

SPRITEPOINTERS:

Navn	Bits	Adresse
SPROPTH	16-31	$DFF120
SPROPTL	0-15	$DFF122
SPR1PTH	16-31	$DFF124
SPR1PTL	0-15	$DFF126
SPR2PTH	16-31	$DFF128
SPR2PTL	0-15	$DFF12A
SPR3PTH	16-31	$DFF12C
SPR3PTL	0-15	$DFF12E
SPR4PTH	16-31	$DFF130
SPR4PTL	0-15	$DFF132
SPR5PTH	16-31	$DFF134
SPR5PTL	0-15	$DFF136
SPR6PTH	16-31	$DFF138
SPR6PTL	0-15	$DFF13A
SPR7PTL	16-31	$DFF13C
SPR7PTL	0-15	$DFF13E

Se på eksempel MC0502 samtidig med at du læser forklaringen:

Linie 1: Lukker alle INTERRUPTs

Linie 3-4: Stopper motoren på alle diskettestationer.

Linie 6: Lukker BITPLANE-, COPPER- og SPRITE-DMA'erne.

Linie 8-17: Sætter en 256 * 320 PIXELs skærm med en BITPLANE (to farver, 1 + baggrund) op.

Linie 19-33: Lægger SPRITE-adresserne ind i COPPER-listen. Læg mærke til at alle SPRITE-dataene ligger defineret efter hinanden i programmet. Linie 32 lægger 68 til, hver gang loopen udføres. Det får registret Dl til at pege på næste SPRITE-data.

Linie 35-40: Lægger adressen på BITMAPen (skærmen) ind i COPPER-listen.

Linie 42-44: Finder adressen på vores COPPER-liste og lægger den ind i COPPER-POINTERen ($DFF080).

Linie 45: Åbner BITPLANE-, COPPER- og SPRITE-DMA'erne igen.

Linie 48-52: Venter (går i loop) indtil elektronstrålen (VIDEOBEAM) når linie 0.

Linie 54: Hopper til rutinen som flytter SPRITEne.

Linie 56-57: Checker om venstre musknap er trykket ned. Hvis ikke, hoppes der tilbage til "wait".

Linie 59: Lukker COPPER-DMA'en.

Linie 61-63: Henter den gamle (forrige) COPPER-listes adresse frem, og lægger den i COPPER-POINTEREN.

Linie 65: Starter COPPER-DMA'en igen.

Linie 67: Starter alle INTERRUPTs igen.

Linie 68: Afslutter programmet (tilbage til K-SEKA eller (CLI).

Linie 71: Her har vi defineret et WORD til at holde OFFSETen i tabellen, som indeholder SPRITEns koordinater.

Linie 73: Her begynder rutinen, som flytter SPRITEne.

Linie 74: Adderer adressen til "movecount" (linie 71), og lægger den i A5.

Linie 75: Henter værdien, som ligger på den adresse, som A5 peger på.

Linie 76: Lægger 4 til værdien i den adresse, som A5 peger på.

Linie 78: Sammenligner værdien, i D5 med 12000.

Linie 79: Hvis D5 er mindre end 12000, hop til "notend".

Linie 81: Læg værdien 0 (CLEAR) ind i D5.

Linie 82: Læg værdien 0 ind i adressen, som A5 peger på (movecount).

Linie 85-86: Læg 0 ind i D1 og D2.

Linie 87: Læg adressen på "movetable" ind i A5.

Linie 88: Læg værdien 15 ind i D3.

Linie 90: Denne variation af MOVE har du sikkert ikke set før. D5 vil i denne instruktion blive lagt til som en OFFSET for A5.

Lad os tage et eksempel:

```
MOVE.W     10(A1),D1
```

Udfører det samme som:

```
MOVE.L     #10,D2
MOVE.W     (A1,D2),D1
```

Som igen udfører det samme som:

```
MOVE.L     #8,D2
MOVE.W     2(A1,D2),D1
```

Vi vil forklare denne variation mere indgående i maskinkodekapitlet i dette brev. Under alle omstændigheder henter den en værdi fra den adresse, som A5 peger på + OFFSET'en i D5, og lægger den i D1 (som er SPRITEns x-position).

Linie 91: Udfører det samme som instruktionen ovenfor, men har desuden en fast OFFSET på 2 og lægger værdien i D2 (som er y-positionen til SPRITEn).

Linie 92: Lægger adressen på "s8" (SPRITE-dataene på SPRITE 8) ind i A1.

Linie 93: Hopper til rutinen "setspr". Det er denne rutine, som sætter SPRITEns nye position.

Linie 95-128: Dette gentages for SPRITE 7 til 1. Det eneste som er anderledes er den faste OFFSET i koordinat-tabellen (movetable). Den bevirker, at vi får en slangelignende effekt (forskydning), når SPRITEne flyttes.

Linie 129: Hopper tilbage til linie 54, og fortsætter med næste instruktion (programlinie 56).

Linie 131: Denne rutine opdaterer SPRITE-positionerne i tabellen over SPRITE-dataene.

Linie 132: Denne variant af MOVE forklares i maskinkodekapitlet senere i dette brev.

Linie 133: Lægger $81 til den værdi, som allerede findes i D1 (som indeholder SPRITEns x-position). Vi skal lægge $81 (129) til for at få position 0 til at være på venstre kant af skærmen.

Linie 134: Lægger $2C (44) til i D2 (som indeholder SPRITEns y-position). Vi lægger $2C til for at få position 0 til at være øverste linie på skærmen.

Linie 135: Lægger værdien 0 ind i D5.

Linie 136: Lægger D2's første 8 BITs (move.10b10) ind på adressen, som A1 peger på.

Linie 137: Kopierer D2's indhold over i D4.

Linie 138: Denne instruktion bliver nærmere forklaret i maskinkode-kapitlet. Det den gør i dette tilfælde, er at skifte D4 8 BITs til højre.

Linie 139: Skifter D4 to BITs til venstre.

Linie 140: Lægger D4 til i D5.

Linie 141: Lægger D3 til i D2.

Linie 142: Lægger BIT 0-7 (move.lOblO) fra D2 ind på adressen, som A1 peger på, + 2 i OFFSET (kaldes også DISPLACEMENT).

Linie 143: Kopierer D2's indhold over i D4.

Linie 144: Skift D4 8 BITs til højre.

Linie 145: Skift D4 1 BIT til venstre.

Linie 146: Læg D4 til i D5.

Linie 147: Kopierer D1's indhold over i D3.

Linie 148: Udfører en logisk AND til D1, således at alle BITs, bortset fra BIT 0, sættes til "0" (BIT 1-31).

Linie 149: Lægger Dl til i D5.

Linie 150: Lægger BIT 0-7 fra D5 ind i adressen, som A1 (+3 i OFFSET) peger på.

Linie 151: Skifter alle BITs i D3 en til højre.

Linie 152: Lægger indholdet af BIT 0-7 i D3 ind i adressen, som A1 (+ 3 i OFFSET) peger på.

Linie 153: Denne instruktion kommer vi tilbage til i kapitlet om maskinkode.

Linie 154: Hop tilbage til den sidste "bsr setspr" og fortsæt med næste instruktion der.

Linie 156-203: Her er vores COPPER-liste defineret. Den første SPRITE-POINTER begynder på programlinie 157 ($DFF000 + $0120=$DFF120). Læg mærke til, at vi i dette program har lagt farveregistrene ind i ; COPPER-listen (programlinie 181-198). Resten af COPPER-listen skulle du være fortrolig med nu.

Linie 206: Her er defineret 10240 BYTEs (eller 2560 LONGWORDs) til skærmhukommelsen (BITMAPen).

Linie 208-359: Her er vores SPRITE-data til alle 8 SPRITEs defineret.

Linie 362: Her har vi defineret en buffer (12400 BYTEs) til at holde koordinaterne på SPRITEne.

For at køre dette eksempel, skal du hente filen "SCREEN" ind i skærmbufferen (også kaldet "screen"), og filen "MOVETABLE" ind i "movetable[11]-bufferen således:

```
SEKA>ri
FILENAME>screen
BEGIN>screen
END>(tryk RETURN eller skriv -1 (logisk END OF FILE))
SEKA>ri
FILENAME>movetab1e
BEGIN>movetable
END>(tryk return eller skriv -1 (logisk END OF FILE))
```

Det er også muligt at lave dine egne "waves" (bølger). Det gøres med et program, som ligger på KURSUSDISKETTE 1. Gør følgende:

BOOT kursusdisketten.
Læg en diskette, som du vil lagre "wave'en" på, i f.eks. "DF1:".

```
1>wavegen df1:mywave
```

Skærmen vil nu blive sort, og maskinen venter på at du trykker på venstre musknap.

Efter at have trykket på musen en gang, begynder "optagelsen".

Flyt nu krydset rundt på skærmen.

Når du er tilfreds med banerne eller mønstret, tryk en gang til på musen og bevægelserne bliver lagret på disketten.

Bemærk at et sekund med bevægelser, bruger 200 BYTEs af dit lager.

Når du er færdig med dette, kan du gå ind i K-SEKA igen. Assembler programmet igen, og indlæs din egen fil i stedet for "movetable"-filen. Læg mærke til at du skal justere værdien på linie 78 i programmet til længden af din egen fil. Det kan også blive nødvendigt at justere BUFFER-størrelsen på linie 362 (hvis din fil er længere end 12400 BYTEs).

MASKINKODE IV

Den første instruktion, vi skal gennemgå, er en lidt speciel variant af MOVE - og den ser således ud:

```
MOVE.W      (A1,D1),D2
```

Denne adresseringsmetode kalder man for register OFFSET (register DISPLACEMENT).
Den fungerer således (se eksemplet): Indholdet i A1 adderes til indholdet i Dl internt i processoren. Hverken A1 eller D1 forandres under denne sammenlægning. Resultatet af additionen bruges som en adresse på den hukommelsescelle, hvorfra data skal hentes. Derefter lægges dataene fra den angivne adresse ind i D2.

Det hele kan skrives på en enklere måde, nemlig:

```
MOVE.W      #$10010,D2
```

Men af og til benyttes tabeller med data, og så kender man ikke altid den eksakte adresse, hvorfra data skal hentes. Så er register OFFSET ideel. Så hvis vi skriver eksemplet ovenfor om, kommer det til at se således ud:

```
MOVE.L      #$100000,A1
MOVE.L      #$10,D1
MOVE.W      (A1,D1),D2
```

Denne instruktion kan også, foruden de to registre, have en fast OFFSET. Den er også vældig god at have, når det drejer sig om at behandle tabeller (lister) med data. Prøv at gætte hvad denne instruktion udfører:

```
MOVE.L      10(A1,D1),D2
```

Fandt du ud af det? Forklaringen kommer her: Adressen, som data skal hentes fra, regnes ud ved at addere A1, D1 og 10 (den faste OFFSET). Altså: A1 + D1 + 10.

Dette er en adresseringsmåde, som er meget god at kunne. Læs derfor afsnittet flere gange, således at du er sikker på, at du forstår, hvad der menes.

De næste instruktioner vi skal se på, virker måske lidt mere indviklede ved første øjekast, men de er helt logiske i deres virkemåde og derfor ikke så vanskelige at forstå, trods alt. De arbejder på BIT-niveau og ser således ud:

LSL og LSR.

De betyder henholdsvis LOGICAL SHIFT LEFT og LOGICAL SHIFT RIGHT.

Eller oversat: Logisk skift til venstre - og logisk skift til højre.

Vi belyser det med et eksempel:

```
MOVE.B      #%00101100,D0
```

D0 (BIT 0-7) indeholder efter denne instruktion er udført det binære tal 00101100.

Derefter udfører vi for eksempel følgende instruktion:

```
LSL.B       #1,D0 (logisk skift af BITs et
                   skridt til venstre)
```

Efter at denne instruktion er udført, ser indholdet i D0 (BIT 0-7) således ud: 01011000 (Binært naturligvis).

Læg mærke til at det kun er første BYTE (BIT 0-7) i D0, som skiftes. Dette fordi vi benyttede "B" i LSL instruktionen. De andre 24 BITs (BIT 8-32) i D0 bliver altså ikke berørt at instruktionen i vores eksempel.

Lad os tage et eksempel til:

```
MOVE.L      #$645A4364,D0
```

D0 vil nu se ud som i figur 1a. (se ark bagest i dette brev). Derefter udfører vi følgende instruktion:

```
LSL.W       #1,D0
```

BIT 0-15 vil nu blive skiftet/flyttet et skridt til venstre som vist i figur 1b (se ark bagest i dette brev). BIT 0 vil få et "0" indført fra højre, mens indholdet i BIT 15 vil "falde ud". Det forsvinder dog ikke helt, men havner i CARRYen. Dette skal vi forklare mere indgående i et senere brev. Resultatet ses i figur IC.

Som du ser bliver kun det første WORD (BIT 0-15) skiftet en BIT til venstre. BIT 16-31 forbliver uberørte, fordi vi brugte "W" i LSL instruktionen. Skift den anden vej - mod højre - (LSR) virker på samme måde. Du kan også skifte/flytte BITene flere skridt, (op til 8), på en gang på denne måde:

```
      LSR      #5,D0
```

Dette var en måde at skrive skift-operationerne på. Der findes tre andre varianter, som vi tager fat på i et senere brev.

BSR - Branch to SubRoutine

Den næste instruktion er BSR (BRANCH TO SUBROUTINE - hop til under-rutine).

Denne instruktion kan sammenlignes med BASIC-kommandoen GOSUB.

Lad os begynde med et eksempel:

```
1  MOVE.L    #5,D0
2  BSR       rutine
3  RTS
4
5  rutine:
6  ADD.L     #1,D0
7  RTS
```

Linie 1: Læg værdien 5 ind i D0.

Linie 2: Hop til "rutine".

Linie 6: Læg #1 til i D0.

Linie 7: Hop tilbage til linie 2, og fortsæt med den næste instruktion.

Linie 3: Afslut programmet.

Hvis du er usikker på det her, så fortsæt alligevel med at læse om STACKen - der kommer nemlig mere om instruktionen BSR.

HVAD ER EN STACK?

I AMIGA (og i alle andre datamaskiner for den sags skyld) er der afsat et område i hukommelsen, som datamaskinen bruger til midlertidig lagringsplads af data, den behøver under kørsel af et program. Blandt andet skal den kunne lagre adresser, når der hoppes fra et sted i et program til et andet, for at kunne komme tilbage igen.

Ordet "STACK" betyder "pibe/skorsten" eller "stabel". Den sidste oversættelse er den bedste i dette tilfælde. Vi skal her forklare dig hvordan en STACK virker og bruges. Til det har vi lavet en figur (se figur 2a, 2b, 2c, 2d og 2e på arket bagest i dette brev.). Se på den, mens vi forklarer begrebet STACK.

STACKen bruges til at gemme værdier/data for en kortere periode under eksekvering af et program. Processoren (MC-68000) bruger også STACKen til at lagre data i - data, som den skal hente frem og viderebearbejde senere i programmet.

Du har måske tænkt over, hvordan processoren kan huske, hvilken adresse den skal hoppe tilbage til, når den hopper til en SUB-rutine (under-rutine)? Det bruger den STACKen til.

Når programmet for eksempel kræver at der hoppes til en SUB-rutine, lagrer/lægger processoren adressen den hopper fra i STACKen. Når SUB-rutinen er udført, henter processoren adressen den skal hoppe til, fra STACKen. Dette sker helt ("automatisk") uden din medvirken.

Du kan også benytte dig af STACKen for at lagre data midlertidigt. Det er dog vigtigt at du ved, hvad du gør, således at når - eller hvis - processoren skal hente en adresse, den har lagret der tidligere, ikke får dine data, som adresse i stedet for. Dit program vil da uværgerligt gå ned.

For at få et visuelt billede af STACKen, kan man forestille sig følgende:

Forestil dig STACKen som et rør, der står oprejst. Nede i røret ligger et antal tallerkener. I bunden af røret kan man tænke sig, at der ligger en springfjeder, som får den øverste tallerken til at ligge helt oppe ved kanten af røret.

Hvis vi lægger en tallerken i røret, vil denne blive liggende øverst. Hvis vi lægger endnu en tallerken i røret, vil den vi lagde der først, blive liggende under denne nye tallerken. Jo flere tallerkner vi lægger i røret, jo længere ned trykkes dem som blev lagt der først. Så langt, så godt.

Nu skal vi prøve at koble dette billede sammen med det som sker i AMIGA når STACKen bruges. Gå tilbage til programeksempel MC0502 og programlinie 132 (som ser således ud)

```
MOVEM.L        D0-D5,-(A7)
```

Denne instruktion vil lægge værdierne, som er i D0, D1, D2, D3, D4 og D5 i STACKen.

Lad os nu se på programlinie 153:

```
MOVEM.L        (A7)+,D0-D5
```

Denne instruktion gør det modsatte af instruktionen i linie 132. Den henter værdierne fra STACKen og lægger dem ind i D0, Dl, D2, D3, D4 og D5 igen.

Hvad er så ideen med dette? Jo, fordi registrene bliver brugt i programrutinen på en sådan måde at værdierne i registrene er forandrede når processoren hopper tilbage til hovedrutinen igen (programlinie 47-57), er det nødvendigt at lagre værdierne i begyndelsen af rutinen - og hente dem tilbage i slutningen af den.

Lad os nu se hvad der sker i AMIGAen når STACKen bliver brugt:

Adresseregistret A7 kaldes også for STACKPOINTER (STACKpeger).

Figur 2a bagest i brevet er en skitse over STACKen. Tallene på venstre side af skitsen repræsenterer adresserne i hukommelsen, mens pilen (SP) på højre side skal være STACKPOINTEREN, som peger på adresse 1000 (A7 indeholder værdien 1000). Forestil dig nu denne instruktion udført:

```
MOVE.L       #10,D0
MOVE.B        D0,-(A7)
```

STACKen vil nu se ud som i figur 2b.

Du har allerede lært hvordan "(A7)+" fungerer (Se BREV III, men du har endnu ikke lært noget om "-(A7)". Den virker på en lidt anden måde end "+" fordi minustegnet står foran parenteserne. Det får processoren til at trække 1,2 eller 4 fra (afhængig af om du specificerer ".B", ".W" eller ".L") i A7 først, og derefter hente indholdet i adressen, som A7 peger på.

Altså, læg værdien 10 (DECIMALT) ind i D0. Træk 1 fra A7, og læg indholdet i D0 ind på adressen, som A7 peger på.

```
MOVE.L #25,D0
MOVE.B D0,-(A7)
```

STACKen vil nu se ud som i figur 2c. Og så fortsætter vi med:

```
CLR.L  D0
MOVE.B (A7)+,D0
```

STACKen vil nu se ud som i figur 2d. DO som vi satte til 0 (CLR), vil nu indeholde 25 (DECIMALT). Derefter udfører vi følgende instruktion:

```
MOVE.B (A7+,D0
```

STACKen vil nu se ud som i figur 2e, og DO som indeholdt 25 vil nu indeholde 10.

MERE OM BSR

Vi vil nu forklare BSR-instruktionen nærmere. Den bruger også STACKen, og som tidligere nævnt sker dette helt automatisk.

```
BSR  rutine

.....
RTS

rutine:

.....
RTS
```

Det der sker, når processoren udfører en BSR, er at den lægger værdien, som ligger i programtælleren (PC, se BREV II), i STACKen, derefter hopper den til "rutine". Når processoren møder en RTS i slutningen af "rutine"-rutinen vil den hente adressen ud igen fra STACKen og lægge den tilbage i programtælleren.

Forestil dig at vi kørte dette program fra K-SEKA:

```
MOVE.L   #5,D0
RTS
```

Først assembler vi programmet.

```
SEKA>a
OPTIONS> <RETURN>
No Errors
SEKA>
```

Derefter starter vi det.

```
SEKA>j
```

Før K-SEKA hopper til begyndelsen af dit program, lægger det en adresse, som peger på et bestemt sted i K-SEKA, i STACKen. Derefter hopper den til dit program.

Når dit program er udført (det afsluttes med en RTS), vil processoren på samme måde som forklaret ovenfor, lægge værdien som ligger i STACKen ind i programtælleren (PC), og du havner tilbage i K-SEKA.

NOTA BENE

Det er vigtigt at du forstår hvordan en STACK virker - princippet gælder for alle datamaskiner. Hvor den lægges i AMIGA, når du tænder for maskinen er ikke godt at vide - den side af sagen ordner operativsystemet i AMIGA aldeles på egen hånd. Sædvanligvis er den på 4000 BYTEs længde, men det kan du øge selv via STACK-kommandoen i CLI.

Læs afsnittene om STACKen om og om igen indtil du er sikker på, at du har forstået princippet. Vi lover dig at det er det værd.

LØSNINGER TIL OPGAVER I BREV IV

Opgave 0402: Information for skærmlinie nummer 4 begynder på adresse $010078

OPGAVER TIL BREV V

Opgave 0501: Hvilken position (x og y) og højde har en SPRITE med disse positionsdata: $EEB5,$1103

Opgave 0502: Hvor bred kan en SPRITE være?

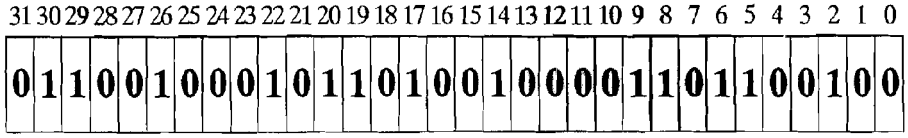
Opgave 0503: Hvor mange SPRITEs er der i AMIGA?

Opgave 0504: D0 = 01100101101001101011011110010111. Hvordan vil D0 se ud efter en "LSR.B #3,D0" ?

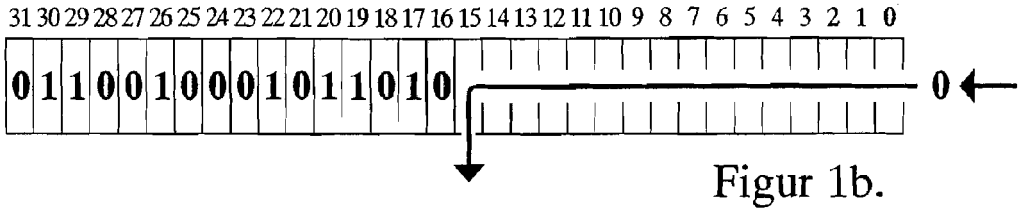
Opgave 0505: Fra hvilken adresse vil D0 få sin værdi i dette programeksempel:

```
MOVE.L    #$150,D1
MOVE.L    #1010,A1
MOVE.B    18(A1,D1),D0
RTS
```

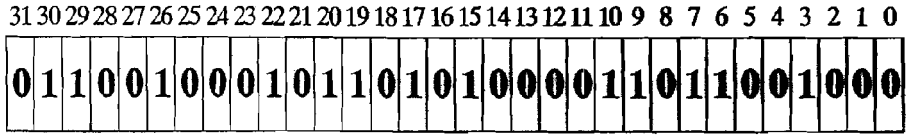
Opgave 0506: Hvad benyttes en STACK til i AMIGA?



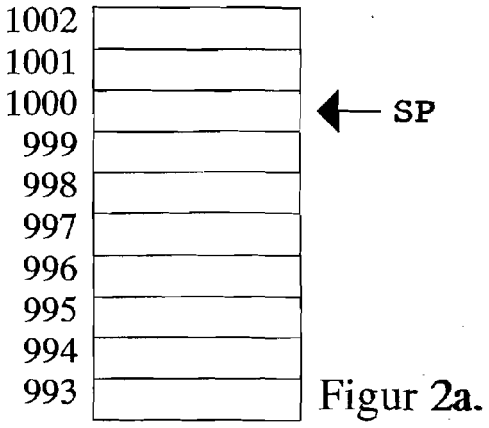
Figur 1a.



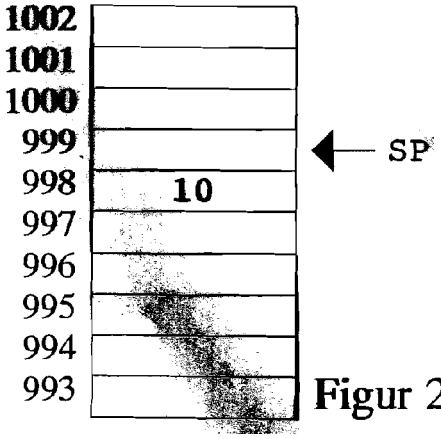
Figur 1b.



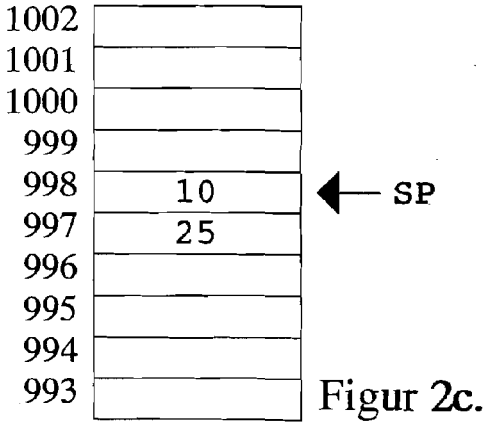
Figur 1c.



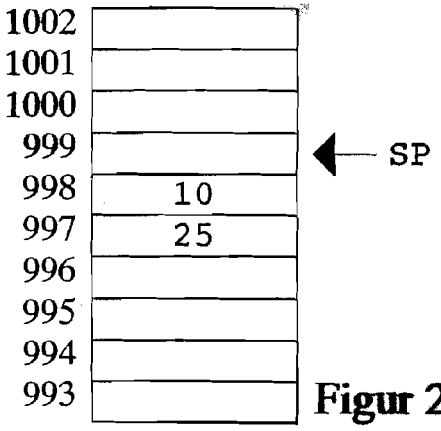
Figur 2a.



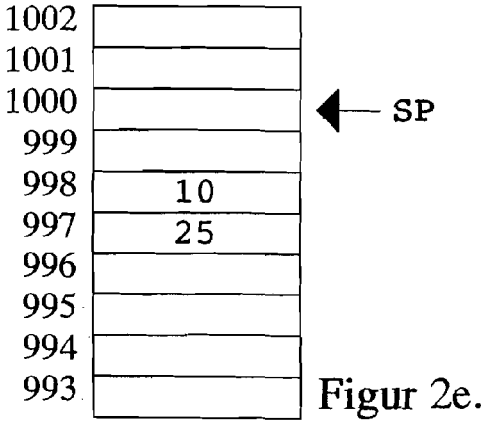
Figur 2b.



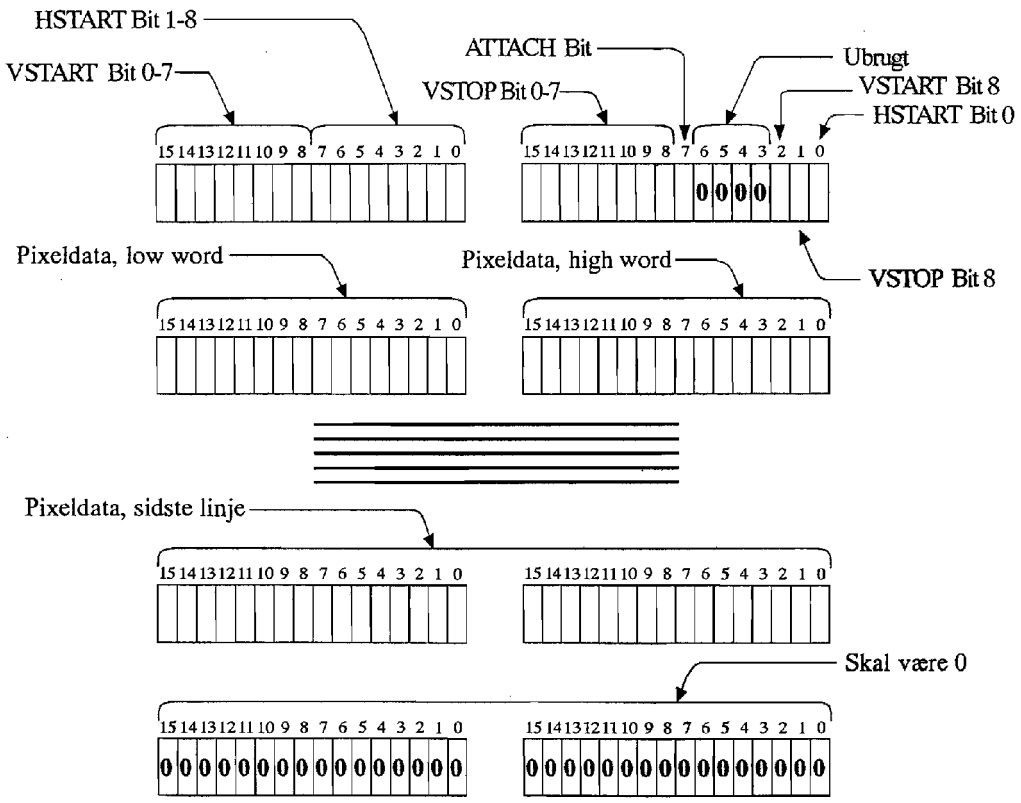
Figur 2c.



Figur 2d.



Figur 2e.


HSTART Bit 1-8
VSTART Bit 0-7
ATTACH Bit
VSTOP Bit 0-7
Ubrugt
VSTART Bit 8
HSTART Bit 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0
VSTOP Bit 8
Pixeldata, low word
Pixeldata, high word
Pixeldata, sidste linje
Skal være 0

Figur 3.