EN DATA BECKER BOG

DULLIN · RETZLAFF
SCHNEIDER · STRASSENBURG

# AMSTRAD

# 464/664 & 6128

# Tips & Tricks

En sand guldgrube for brugere af 464/664 & 6128

# DANSK / NORSK

udgave

Forlag: DE

# TIPS & TRICKS

***EN DATA BECKER BOG***

**DULLIN · RETZLAFF**
**SCHNEIDER · STRASSENBURG**

# AMSTRAD

# 464/664 & 6128

# Tips & Tricks

**En sand guldgrube for brugere af 464/664 & 6128**

# DANSK / NORSK

**udgave**

NORDIC COMPUTER SOFTWARE
SMEDEGADE · POSTBOX 105 · DK-6950 RINGKØBING
1986

Postbox 105
Smedegade 7
DK-6950 Ringkøbing

Distribution i Norge:
Computer Equipment A/S
Gyldenløves gt. 42
N-4600 Kristiansand S.
Tlf. (042) 70 294

Sats & tryk: Tarm Bogtryk & Offset A/S

Dansk oversættelse & bearbejdning: Peter Friis

ISBN 87-7283-009-3



## VIGTIGT

Programmer, programeksempler mv. er omfattet af lov om copyright. Disse må kun anvendes til personlige- eller undervisningsformål og må ikke anvendes kommercielt.

Alle programeksempler, tekniske anvisninger osv. i denne bog er omhyggeligt gennemgået for fejl. Trods dette kan der optræde fejl i reproduktionsfasen.
Skulle nogle læsere opdage fejl, beder vi Dem rette henvendelse til os, så vi har mulighed for at foretage rettelse.

# FORORD

Kært barn har mange navne!

De i denne bog omtalte computere 464, 664 & 6128 kendes under flere navne, men her er det nok med de 2, nemlig AMSTRAD og CPC.

Når begge navne i denne bog bruges i flæng, er det netop for at gøre læserern fortrolig med, at begge navne bruges om samme computer.

De fleste TIPS & TRICKS gælder for alle tre computere (464, 664/6128), men der er undtagelser, se kapitlerne 16-20 og 21.

Denne bog kan godt "stå" alene, men visse ting vil måske nok være lettere at forstå, hvis man først læser bogen "PEEKS & POKES".

Med ønsket om at bogen må bringe læseren meget mere udbytte af computeren, så

*God fornøjelse.*

# INDLEDNING

## DENNE BOGS INDDELING

Denne bog er i inddelt i tre afsnit med hver sin overskrift:

Bogens første del beskæftiger sig udelukkende med AMSTRAD'ens BASIC. Der vil blive vist og gennemgået en del nyttige BASIC-programmer og rutiner, samt en række tips til programmering i BASIC. I dette afsnits slutning findes en komplet beskrivelse af AMSTRAD-specifikke BASIC-kommandoer.

Anden del omhandler maskinkoderutiner, der kan supplere og udvide BASIC-fortolkerens kunnen. Selvom programmerne i dette afsnit er maskinkodeprogrammer "READY-TO-GO", så kan de også bruges af læsere, der måske ikke ved så meget om den slags programmering ..... endnu.

Selv om programmerne er lige til at indtaste, så er der bestemt også både tips og nyt at hente i afsnittet.

Bogens tredie del beskæftiger sig med Amstrad computerens maskinsprog. Afsnittet indeholder en lang række interessante rutiner fra operativsystemet og BASIC-interpreteren (oversætteren). Der er også en række tips og tricks til en mere effektiv (tidsmæssig) programmering i maskinsprog både hvad angår operativsystem og f.eks. kommandoudvidelse med RSX. Afsnittets kapitel 23 omhandler en metode til relokalisering (forskydning med tilspasning til absolutte adresser) af maskinkodeprogrammmer. Metoden er, såvidt vides, ikke tidligere blevet offentliggjort. Kapitlet er gjort særligt udførligt, for virkelig at vise, hvordan man kan programmere ved trickmæssig behandling af Stack. Mon ikke dette kapitel kan lære selv en øvet maskinsprogsprogrammør lidt mere om. Selvom bogens tredie del næsten udelukkende handler om maskinsprogsprogrammering, indeholder det også stof for den programmør, der indtil nu ikke har beskæftiget sig med den form for programmering. Især de to første kapitler i afsnittet er en slags indføring i denne form for programmering og indeholder derfor talrige eksempler.

## DE HEXADECIMALE TALS NOTATION

De hexadecimale tal i denne bog er hovedsaglig angivet med et foranstillet "&" . Det er den samme notation som computeren bruger i BASIC. Nogle ASSEMBLERE benytter andre slags notationer; f.eks. et "#" eller et efterstillet "H". Benytter man selv en af disse assemblere, så må man naturligvis konvertere til dennes notation.

## SPECIALTEGN I LISTNINGER

I denne bog vil læseren ofte støde på tegnet "↑". Der er ingen grund til at fortvivle selvom tegnet ikke findes på computerens tastatur. Tegnet svarer til computerens

opretstående pil og tjener som potensoperator. Normalt vil potenstegnet resultere i et "↑" på en printer.

## CPC'S TASTATUR

Uheldigvis er ikke alle taster på de tre CPC'ere ens. Hvor der i denne bog skrives RETURN (som på 6128), menes der på 664 og 464 den store ENTER-tast i hovedblokken.

# INDHOLDSFORTEGNELSE

## FØRSTE DEL
## TIPS & TRICKS TIL BASIC



## ANDEN DEL
## KOMMANDOUDVIDELSER OG ANDRE NYTTIGE MASKINKODEPROGRAMMER



## TREDIE DEL
## TIPS & TRICKS TIL MASKINSPROG

# FØRSTE DEL

# TIPS & TRICKS TIL BASIC

# 1. SORTERING

Hvorfor nu dette underlige tema?

Særlig interessant er undersøgelsen af de forskellige sorteringsmuligheders hastigheder. En stor del af en computers arbejdstid går med at sortere. Der findes ikke ret mange brugerprogrammer, der ikke gør brug af sortering. Det er altså ikke uden grund, at man løbende forsøger at finde på nye og mere effektive (hurtigere) sorteringsrutiner.

Oprindeligt, skulle dette kapitel ikke have været særlig omfangsrigt, men ved intensiv beskæftigelse med dette tema, viste det sig at være så spændende, at vi har valgt at uddybe emnet.

Sorteringsrutiner er ofte fyldt med alskens tricks, der programmeringsmæssigt ikke fylder så meget. Det er især de enklere programmer på omkring 10 linier, der kan fremvise forbavsende præstationsforskelle. Det vanskelige ligger altså ikke i at skrive sit program i et eller andet sprog, men i at gøre programmet så effektivt som muligt.

Ved sortering er der to faktorer, der især spiller ind, når det drejer sig om regnetid:

1. Det gennemsnitlige antal sammenligninger, der skal udføres.

2. Det gennemsnitlige antal data-forskydninger og data-udskiftninger.

Det er disse to faktorer, der skal holdes på et så lille antal som muligt.

En algoritmes kvalitet består af disse størrelser. Problemet er at sorteringstiden ikke er proportional med mængden af data, der skal sorteres. Det betyder:

Når det tager 1 sekund at sortere 10 navne, så fordobles tiden ikke nødvendigvis, dersom der skal sorteres 20 navne. Faktisk drejer det sig i mange tilfælde om kvadratiske tidsværdier, når der anvendes dårlige rutiner. Ved det dobbelte antal data, skal man bruge 4 gange så lang tid (4=2↑2) og ved 10 gange mængden af data, stiger tiden til faktor 100 (100=10↑2).

Den såkaldte "BUBBLE SORT" har fået sit navn på grund af metoden, der lader de større elementer stige op som luftbobler i vand. På denne måde har man til sidst et sorteret felt. Elementerne sammenlignes parvis. Hvis et element med et mindre indeks er større, vil de to elementer bytte plads. Den samme sammenligning sker med det næste par o.s.v. Hele proceduren er først afsluttet, når hele feltet er gennemarbejdet og de største elementer befinder sig øverst.

Hvis et felt består af n elementer, skal der ialt sorteres n-1 gange.

Når det til enhver tid største element er bragt på plads foroven, skal der ikke længere tages hensyn til det. Med andre ord, så lader man som om, der startes på en helt ny

sortering. Efter det andet gennemløb er det største element blevet fundet og bragt på plads foroven. På det tidspunkt er der foretaget (n-1)+(n-2) sammenligninger. Når sorteringen er tilendebragt vil der være foretaget

(n-1)+(n-2)+(n-3)+.....+2+2=n*(K-1)/2

sammenligninger. Hvis n=10 skal der bruges 45 sammenligninger, for n=100 bliver det 4950, hvilket vil sige mere end 100 gange så mange sammenligninger når antallet er 10 gange så stort. Sortertiden stiger kvadratisk. For nedenstående programeksempler gælder følgende:

1. Feltet, der skal sorteres befinder sig i tabellen a(ant).

2. Det højeste indeks befinder sig i "ant". Således bliver antallet af elementer, der skal sorteres ant+1, idet der også ligger et element i indeks 0.

3. Efter sortering befinder det sorterede felt sig stadigvæk i a(ant).

4. Alle variabler, der ikke er kendetegnet med $ eller !, betragtes som heltalsvariabler (integer) af tidsmæssige grunde.

Her følger grundprogrammet, der sørger for de indledende procedurer:

```
 10 ' Sortering
 20 DEFINT a-z:DEFREAL t
 30 RANDOMIZE TIME
 40 ud=0:'Flag for udskrivning af sortering
 50 udf=0:'Filnummer for udskrivning
 60 ant=20:ant=ant-1:'Feltets størrelse
 70 DIM a(ant+1),b(ant),l(20),r(20)
 80 For i=0 TO ant:a(i)=INT(100*RND):b(i)=a(i):NEXT
 90 READ j$:IF j$="#" THEN END ELSE j=VAL(j$):PRINT j,:IF ud
    THEN PRINT
100 FOR i=0 TO ant:a(i)=b(i):NEXT:t=TIME
110 ON j GOSUB 200,300,400,500,600,700,800,900,1050,1200
120 PRINT#udf,(TIME-t)/300
130 GOTO 90:'Slutmarkering
140 DATA 1,2,3,4,5,6,7,8,9,10
150 DATA #
160 REM Subrutine for udskrivning af sortering
170 FOR n=0 TO ant:PRINT #udf,USING"##";a(n);:
    PRINT #udf,"  ";:NEXT
180 PRINT#udf
190 RETURN
```

Men nu til BUBBLE SORT:

```
200 REM BUBBLESORT
210 FOR i=ant TO 1 STEP -1
220 FOR j=1 TO i
230 IF a(j-1)>a(j) THEN s=a(j-1):a(j-1)=a(j):a(j)=s
240 If ud THEN GOSUB 170
250 NEXT j,i
260 RETURN
```

Den yderste løkke formindsker antallet af data, der skal sorteres i feltet med 1 for hvert gennemløb. I den inderste løkke sammenlignes restelementerne parvis og ombyttes evt.

For at få de korrekte tidsværdier, skal linie 240 fjernes fra programmet. Det er den linie, der tjener til udskrivning af feltet efter hver sammenligning. Udskrivningen fremkommer ved at sætte variablen til -1. Man kan således følge hele sorterproceduren fra første til sidste element.

Her er nogle forslag til udvidelser:

Indbyg en tæller for antallet af sammenligninger og en for antallet af ombytninger. Sammenhold værdierne med tiden og de teoretiske værdier.

## BUBBLESORT ER EN DÅRLIG SORTERINGSMETODE!

Dette eksempel viser hvor små ændringer, der skal til for at forbedre en algoritme betydeligt. En af Bubblesort's svagheder er, at den stadigvæk sorterer, når alle elementer er placeret på rette plads. Det kan vi få bekræftet ved at sætte et flag, når der sker en ombytning. Hvis flaget endnu ikke er blevet sat efter et gennemløb, og der altså ikke er sket nogen ombytning, er sorteringen færdig. Prøv at regne ud, hvor meget tid (antal gennemløb og ombytninger), der er sparet.

Ideen med at sætte et flag kan videreføres. I stedet for at sætte flaget, kan vi gemme det indeks, hvori ombytningen er sket. Efter et gennemløb vil flaget få værdien af indekset for den sidste ombytning, hvilket vil sige, at alle elementer med et højere indeks er færdigsorterede. Således er det spild af tid at løbe hele rækken igennem hver gang. Vi behøver kun at lade programmet løbe til det lagrede indeks:

```
300 REM udvidet BUBBLESORT
310 FOR i=ant TO 1 STEP -1
320 maxj=0
330 FOR j=1 TO i
340 IF a(j-1)>a(j) THEN s=a(j-1):a(j-1)=a(j):a(j)=s:maxj=j
350 IF ud THEN GOSUB 170
360 NEXT j
370 IF maxj=0 THEN RETURN ELSE i=maxj
380 NEXT i
390 RETURN
```

Prøv at undersøge, hvor meget forbedringen betyder for programmet. Det ville også være en ide at skrive en Bubble-version, hvor der skiftevis "bobles opefter og nedefter", d.v.s. at de største elementer stiger op, og de mindste stiger nedefter. Benyt to pointere til at "indhegne" det område, der endnu ikke er blevet sorteret.

Nok om bobler! Nu følger et par simple sorteringsrutiner med store individuelle forskelle. Udfør iøvrigt de samme tests på disse algoritmer.

MAXSORT

Maxsort undersøger det endnu ikke sorterede felt for det største element. Findes der et element, der er større end det, som er lagret som det sidste, så byttes de to om. For hvert gennemløb indskrænkes feltet.

```
400 REM Maxsort
410 FOR i=ant TO 1 STEP -1
420 FOR j=0 TO i-1
430 IF a(j)>a(i) THEN s=a(j):a(j)=a(i):a(i)=s
440 IF ud THEN GOSUB 170
450 NEXT j,i
460 RETURN
```

STRAIGHT-SORT eller sortering ved udvælgelse

Fremgangsmåden er i princippet en forbedret udgave af maxsort.

Forbedringen ligger i, at der ikke automatisk byttes ved møde med et større element, men derimod findes det største i restfeltet.

Antallet af sammenligninger svarer til det formindskede antal ombytninger.

```
500 REM straight
510 FOR i=ant TO 1 STEP -1:m=0
520 FOR j=0 TO i
530 IF a(j)>m THEN m=a(j):k=j
540 NEXT
550 a=a(i):a(i)=a(k):a(k)=a
560 IF ud THEN GOSUB 170
570 NEXT
580 RETURN
```

INSERT-SORT eller sortering ved indføring

Insert-Sort gør brug af en metode, vi allesammen kender. Ved kortspil sorteres hvert nyt kort ind i "hånden" ved indføring blandt de andre kort. Det er nøjagtigt på samme måde, rutinen arbejder:

Det nye kort indlemmes midlertidigt, som det største kort. Derefter sammenlignes det med sine naboer. Hvis det er mindre, så byttes der plads.

```
600 REM InsertSort
610 FOR i=1 TO ant
620 FOR j=i TO 1 STEP -1
630 IF a(j)>=a(j-1) THEN 670
640 s=a(j-1):a(j-1)=a(j):a(j)=s
650 IF ud THEN GOSUB 170
660 NEXT j
670 NEXT i
680 RETURN
```

Da en forskydning er bedre end en ombytning, så kan Insert-Sort forbedres i retning af Maxsort:

Først findes det sted, hvor det nye element skal indsættes. Så forskydes alle større elementer for at give plads til det nye.

```
700 REM forbedret InsertSort
710 FOR i=1 TO ant
720 a=a(i)
730 FOR j=i-1 TO 0 STEP -1
740 IF a<a(j) THEN NEXT j
750 FOR k=i TO j+2 STEP -1:a(k)=a(k-1):NEXT k
760 a(j+1)=a
770 IF ud THEN GOSUB 170
780 NEXT i
790 RETURN
```

Men selv denne version kan ændres. Forskydning af feltet kan ved mindre felter, gøres mere effektiv med en ekstra løkke.

```
800 REM 3. version InsertSort
810 FOR i=1 TO ant
820 a=a(i):j=j-1
830 IF a>=a(j) THEN 860
840 a(j+1)=a(j):j=j-1
850 IF j>=0 THEN 830
860 a(j+1)=a
870 IF ud THEN GOSUB 170
880 NEXT
890 RETURN
```

Binary Sort = Binær søgning med InsertSort

I det ovennævnte program blev en stor del af programtiden brugt til at finde en ny plads til elementet i et allerede sorteret felt. Den kostbare tid kan nedsættes betydeligt ved brug af Binary Search. Ved normal InsertSort skal hvert element trinvis undersøges, hvor man med Binary Sort bruger lidt "intelligens". Antag at man skal finde et bestemt tal mellem 1 og 128. Efter InsertSort-metoden ville man gå således frem:

Er tallet 128?
Nej!
Er tallet 127?
Nej!
Er tallet 126?
-
-
-

Det ville være betydeligt mere effektivt at spørge således:

Er tallet større, mindre eller lig med 64?
Større en 64!

Næste spørgsmål lyder så:

Er tallet større, mindre eller lig med 96?
Mindre end 96!

Så:

Er tallet større, mindre eller lig med 84?
Mindre end 84!

Princippet i denne fremgangsmåde er indkredsning af det søgte tal. Går man systematisk frem, er antallet af mulige svar begrænset.

En lignende måde at bruge teknikken på, er set i en kendt tv-quiz:

Er personen han- eller hunkøn ?

De mulige svar vil naturligvis være henholdsvis hankøn og hunkøn.

Intervallet, hvori et tal kan befinde sig, halveres for hvert spørgsmål. Tallet, der skal findes er:

30

Intervallet efter hvert spørgsmål er da:

| SPØRGSMÅL | SVAR | INTERVAL |
|---|---|---|
| >,< eller = 64 | <64 | 1-63 |
| >,< eller = 32 | <32 | 1-31 |
| >,< eller = 16 | >16 | 17-31 |
| >,< eller = 24 | >24 | 25-31 |
| >,< eller = 28 | >28 | |
| >,< eller = 30 | = 30 | FUNDET!!!! |

Der var brug for 6 spørgsmål (sammenligninger) for at finde det korrekte tal. Efter den gamle metode, skulle der udføres 98 sammenligninger. Det maksimale antal sammenligninger ved n-elementer ved binær søgning, er proportional med:

$ln_2 2$ ($ln_2$ = logaritmen til basis 2)

Fremgangsmådens navn skyldes at intervallet halveres for hvert spørgsmål. Det er derfor, vi bruger logaritmen til basis 2. Tidsbesparelsen er rent ud sagt enorm ved store mængder data. Et elements position blandt 60 millioner elementer, kan findes ved blot 26 sammenligninger. Prøv at gennemgå programmet og forestil dig, hvad der sker for hver instruktion. På grund af den lidt mere komplicerede programstruktur, opnår man først hastighedsfordele ved sortering af mere end 100 elementer.

```
 900 REM BinarySort
 910 FOR i=1 TO ant
 920 IF a(i)>a(i-1) THEN 1000
 930 l=-1:r=i+1
 940 h=INT((l+r)/2)
 950 IF a(i)>a(h) THEN l=h ELSE r=h
 960 IF r>l+1 THEN 940
 970 a=a(i):FOR j=i TO r+1 STEP -1: a(j)=a(j-1):NEXT
 980 a(r)=a
 990 IF ud THEN GOSUB 170
1000 NEXT
1010 RETURN
```

SHELLSORT

Denne rutine har fået navn efter sin kunstruktør D.L. Shell, der udviklede den i 50'erne. Det interessante er, at rutinen ligner Bubble-Sort til forveksling; bortset fra den lille men vigtige forskel, at feltet deles op i et antal mindre underfelter, der sorteres og efterhånden udgør hele feltet.

Ideen er at feltet først "grovsorteres". Hvis vi eksempelvis har 16 elementer, så sammenlignes først 1. og 9. element, så 2. og 10., 3. og 11., o.s.v. Der byttes om efter behov. Dermed er grovsorteringen udført. Ved dette gennemløb, blev elementer med afstanden 8 sammenlignet.

Ved den næste gennemgang er sammenligningsafstanden halveret til 4. Nu sorteres underfelterne, der gives med afstanden på 4 elementer, d.s.v. afstandene mellem 1. og 5., 9. 13. så 2. og 6., 10. og 14. o.s.v. De nye underfelter sorteres herefter efter Insert-metoden.

Efter hver gennemgang halveres sammenligningsafstanden indtil tallet 1 er nået. Den sidste sortering foregår i hele feltet.

Sheel-rutinen er en meget interessant algoritme, der er de indtil nu gennemgåede rutiner langt overlegen, sålænge det drejer sig om felter med over 20 elementer.

```
1050 REM ShellSort
1060 FOR m=ant-1 TO 1 STEP -1
1070 m=INT((m+1)/2)
1080 FOR j=0 TO ant-m
1090 i=j
1100 IF a(i)<=a(i+m) THEN 1150
1110 s=a(i):a(i)=a(i+m):a(i+m)=s
1120 IF ud THEN GOSUB 170
1130 i=i-m
1140 IF i>=0 THEN 1100 ELSE 1150
1150 NEXT j,m
1160 RETURN
```

## QUICKSORT

En af de hurtigste sorteringsrutiner er Quick-sort. Hvor det drejer sig om felter med mere end 50 elementer, er det langt den hurtigste af de rutiner, vi har gennemgået indtil nu. Princippet i Quick-sort ser således ud:

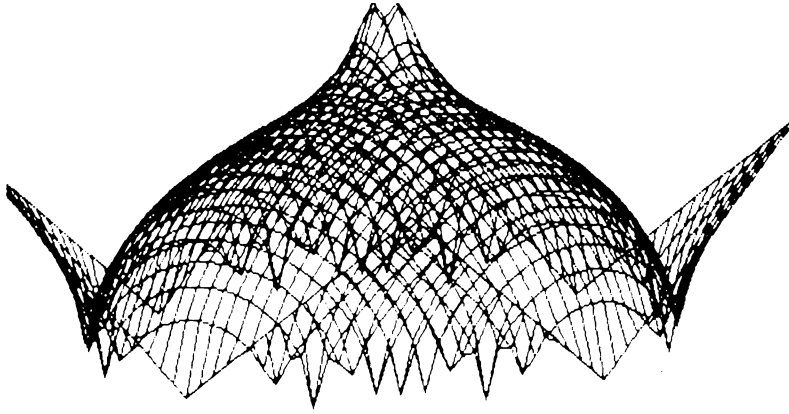
Feltet, der skal sorteres deles op i to halvdele. Elementet mellem de to halvdele tjener som reference-element. Nu bliver den nederste halvdel af de elementer, som er større end referenceelementet byttet om. I nogle tilfælde udskiftes elementet, der sammenlignes med. Denne første sortering svarer til en udvidet udgave af grovsorteringen i Shell-rutinen.

Efter det første gennemløb, gentages proceduren med hver af de to halvdele. D.v.s. at proceduren er rekursiv (kalder sig selv). For at kunne lave denne rekursive struktur (den stammer fra Pascal) i BASIC, skal de vigtigste variabler gemmes inden gentagelsen, da deres indhold ellers går tabt. Dertil bruges tabellerne l(sp) og r(sp).

```
1200 REM QuickSort
1210 sp=0:l(sp)=0:r(sp)=ant
1220 l=l(sp):r=r(sp):sp=sp-1
1230 i=l:j=r:vv=a(INT((l+r)/2))
1240 WHILE a(i)<vv :i=i+1:WEND
```

```
1250 WHILE a(j)>vv :j=j-1:WEND
1260 IF i>j THEN 1300
1270 s=a(i):a(i)=a(j):a(j)=s:IF ud THEN GOSUB 170
1280 i=i+1:j=j-1
1290 IF i<=j THEN 1240
1300 If i<r THEN sp=sp+1:l(sp)=i:r(sp)=r
1310 r=j:IF l<r THEN 1230
1320 IF sp>=0 THEN 1220
1330 RETURN
```

# 2. 3-D-GRAFIK

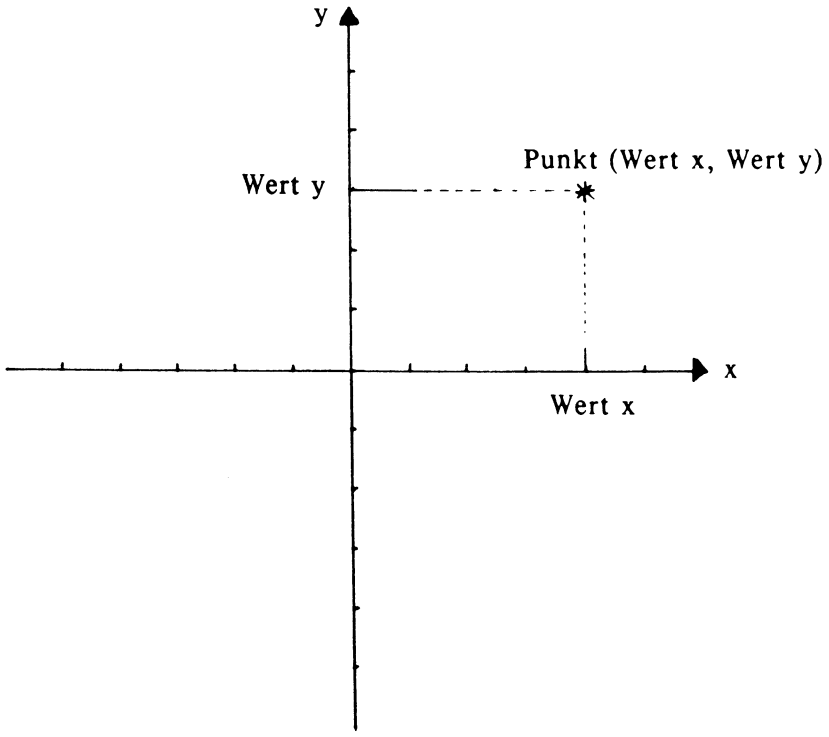


```
40 DEF FNf(x)=SQR(ABS((16-x*x-z*z)))+1/SQR(x*x+z*z+0.1)
50 ap=34
80 DATA -4,4,-2,6,-4,4
```

En af de flotteste og mest interessante anvendelser af computeren, er opbygningen af grafik og især 3-D-grafik. Den komplette teori omkring 3-D-grafik er særdeles komplex og vidt udbygget i takt med udviklingen indenfor CAD (Computer Aided Design). Med de nyeste processor-generationer, er det endda blevet muligt at lade en computer skabe og udføre levende (animerede) billeder til brug i film og simulatorer.

Så langt kan vi dog ikke gå i denne bog. Men Amstrad-computerne er ikke tabt bag af en vogn, når det drejer sig om grafikmuligheder. MODE 2's opløsning på 640*200 punkter er ideel til opbygning af tredimensionale funktioner.

En funktion kan vises i en værditabel, hvori hver af værdierne fra definitionsmængden er tilknyttet en funktionsværdi. Det fremkomne talpar kan vises grafisk i et koordinatsystem. De to tal kaldes også for "punktets koordinater". Se f.eks. den følgende funktion:




Her er nogle talpar fra funktionen:

| X | Y |
|---|---|
| 1,5 | 1 |
| 3,7593 | 3 |
| 1000,379 | 1000 |
| 2 | 2 |
| -1,5 | 1 |

Vi vil nu lade computeren tegne denne funktion. Dermed opstår det næste problem: Hvordan fortæller man en sådan opgave forståelig (i matematisk form) for en computer?

I dette eksempel, er det nemt. Den ovennævnte funktion fremkommer i computeren ved hjælp af INT-funktionen. Værdierne for definitionsmængden ligger i variablen X. Værdimængden ligger i Y. Funktionen skal vises i intervallet -5 til 15. Et første forsøg kunne se således ud:

```
10 MODE 2
20 FOR X=-5 TO 15
30 Y=INT(X)
40 PLOT X,4
50 NEXT
```

Men vi opnår ikke den ønskede effekt på denne måde. Programmet skal først tilpasses computerens skærmformat. Skærmens bredde svarer til 640 enkelte punkter på række (X-aksen). Højden er opdelt i 200 punkter (Y-aksen), der også kan betegnes med værdierne fra 0 til 400 i et koordinatsystem. (Halveres internt).

Billedet, der fremkom ved vort eksempel, er en miniature af det ønskede format. De tilsvarende X- og Y-værdier skal så at sige "strækkes". Det bliver nødvendigt at indføre nogle målestoksfaktorer, der skal "presse" eller forlænge de reelle værdier for X og Y på en måde, så skærmen udnyttes bedst muligt.

Funktionen skal fremstå i intervallet fra XOBG=15 til XUNTG=-5. Det samlede område, der skal vises, får størrelsen XOBG-XUNTG=15-(-5)=20. Da der er 640 punkter til rådighed, skal de 20 forlænges til 640, hvilket vil sige at X-værdierne skal multipliceres med 640/20=32.

```
10 MODE 2
20 XUNTG=-5:XOBG=15
30 MASX=640/(XOBG-XUNTG)
40 FOR X=XUNTG TO XOBG STEP 1/MASX
50 Y=INT(X)
60 PLOT X*MASX,Y
70 NEXT X
```

Step-funktionen gør, at der sættes et punkt for hver af de 640 X-værdier. Det fremkomne billede er endnu lidt kedeligt, idet Y-værdierne ikke er blevet "strakt".

Som ved X-aksen, skal vi også ved Y-aksen vide i hvilket interval, værdierne må ligge. Det er ikke altid lige let at finde frem til, når man har med en ukendt funktion at gøre. Enten, må man prøve sig frem, eller også må man lave en testkørsel, hvor man indsætter de maksimale værdier for X og Y. Disse værdier benyttes så, som yderpunkter for de ønskede grafer.

På grund af vores funktion's egenskaber, er det nemt at se, at Y vil bevæge sig indenfor intervallet -5 til 15. Tilføj disse linier eller foretag ændringen i de eksisterende.

```
21 YUNTG=-5:YOBG=15
31 MASY=400/(YOBG-YUNTG)
```

Linie 60 skal ændres til:

```
60 PLOT X*MASX,Y*MASY
```

Til trods for indførelsen af målestoksfaktorer, så udnytter grafen ikke hele skærmbilledet. Det er nødvendigt på baggrund af X og Y, at finde over- og undergrænse for skærmens nulpunkt. Normalt ligger dette punkt i nederste venstre hjørne. Rent principielt, kunne vi fastlægge nulpunktet (på engelsk: ORIGIN) med den dertil beregnede kommando ORIGIN:

```
35 ORIGIN 160,100
```

Denne metode, der naturligvis er den enkleste, fungerer imidlertid kun, hvis nulpunktet også bogstavligt befinder sig på det angivne skærmkoordinat. Det nytter således ikke at bruge

```
ORIGIN -32000,-20000
```

hvis et stykke af en funktion (2000 til 2040) skal vises. Det vil afstedkomme fejlmeddelelsen:

Overflow

De nødvendige beregninger skal derfor udføres i selve programmet. Det gøres ved at forskyde billedskærmen i X- og Y-retning. Følgende ændringer til linie 60 sørger for dette. Slet linie 35 og indtast ORIGIN 0,0.

```
60 PLOT (X-XUNT)*MASX,(Y-YUNTG)*MASY
```

Tilføjer vi følgende linier, får vi også akserne på skærmen, hvis de ligger der.

```
35 IF XUNTG*XOBG>0 THEN 37
36 MOVE(X-XUNTG)*MASX,0:DRAW (X-XUNTG)*MASX,400
37 IF YUNTG*YOBG>0 THEN 40
38 MOVE 0,(Y-YUNTG)*MASY:DRAW 640,(Y-YUNTG)*MASY
```

Den netop viste funktion INT(X) har en mindre pæn egenskab, idet den tegnede graf forløber i "knæk" fra punkt til punkt i stedet for pæne regelmæssige streger og buer. I det følgende vil vi kun se på "pæne" og regelmæssige funktioner. En af de nemmeste funktioner er den, hvor X afbilleder sig selv. Som det er nemt at efterprøve, så giver dette en ret linie, der forløber diagonalt i koordinatsystemet. Som eksempel på en simpel regelmæssig funktion, vil vi kigge på en kvadrat, hvis funktion ser således ud:

$y = x^2$

Først defineres funktionen med kommandoen DEF FN:

```
15 DEF FNY(X)=X↑2
```

ændring

```
50 Y=FNY(X)
```

Da en funktion er temmelig stabil, d.v.s. at den ikke afviger ret meget over et relativt kort stykke på X-aksen, så er det muligt at se bort fra metoden med at sætte enkeltpunkter. Man kan i stedet for sætte nogle punkter i rimelig afstand, og bagefter "trække" linierne mellem punkterne op. Alt efter afstanden opnås en tidsgevinst på fra 10 til 50 gange. Det er netop denne kendsgerning, der gør 3-D-netgrafikprogrammet i slutningen af dette kapitel så hurtig. Derfor skal der foretages følgende ændringer i vores nuværende program:

```
16 PUAB=10:REM punktafstand
```

Linie 40 skal ændres til linie 42.

```
40 MOVE 0,(FNY(XUNTG)-YUNTG)*MASY
42 FOR X=XUNTG TO XOBG STEP 1/MASX*PUAB
```

Linie 60 ændres til:

```
60 DRAW (X-XUNTG)*MASX,(Y-YUNTG)+MASY
```

Yderligere skal Y-grænserne p.g.a. den nye funktion ændres til -10 og 250.

Prøv forskellige andre kvadratiske funktioner:

| FUNKTION | XUNTG | XOBG | YUNTG | YOBG |
|---|---|---|---|---|
| $X^2$-50 | -10 | 15 | -150 | 200 |
| $X^2$+20 | - 5 | 10 | - 10 | 90 |
| $(X-5)^2$ | - 5 | 15 | - 10 | 100 |
| $(X+5)^2$ | -10 | 10 | - 10 | 250 |
| $(X-5)^2$-50 | - 5 | 15 | - 60 | 60 |

Som det forhåbentligt står klart for læseren, er fremstilling af to-dimensionale funktioner relativt simpel. En to-dimensional funktion ligger i eet plan, og er derfor nem at flytte over på en skærm. Når det drejer sig om tre-dimensionale funktioner, så er problemet, *at en skærm kun kan indeholde to dimensioner.*

Ved en tre-dimensional grafisk afbildning bliver hvert talpar i definitionsmængden tilordnet en værdi i værdimængden. F.eks., kunne man forestille sig, at definitionen går på et bord, hvor hvert punkt i bordet er tilordnet en bestemt højde i forhold til bordpladen. For at skabe en "ægte" afbildning i tre dimensioner, måtte fladerne, der skal afbildes, formes i et medgørligt materiale som f.eks. "gumminet".

For programmøren består opgaven i at kunne omdanne en 3-dimensional figur til skærmens 2-dimensionale format. Skal fremstillingen være virkelighedsnær, så må de dele af billedet, som ligger længst "borte" være mindre og omvendt. Denne effekt giver et indtryk af et virkeligt perspektiv i billedet. Når det drejer sig om grafisk fremstilling af funktioner, kan vi se bort fra denne mulighed.

Lad os se på denne funktion:

```
DEF FNY(X)=X↑2+Z↑2
```

hvor Z indeholder værdien for den 3. akse. I vores model svarer den til bordets korte side. Hvis vi antager, at den 2. linie indeholder punktet -1 på Z-aksen, eller sagt på en anden måde; at den har afstanden 1 bagud fra den 1. linie. Linien tegnes op ved, at Z sættes til -1 og den normale løkke for linien gennemløbes. Dernæst sættes Z til værdien for den næste linie.

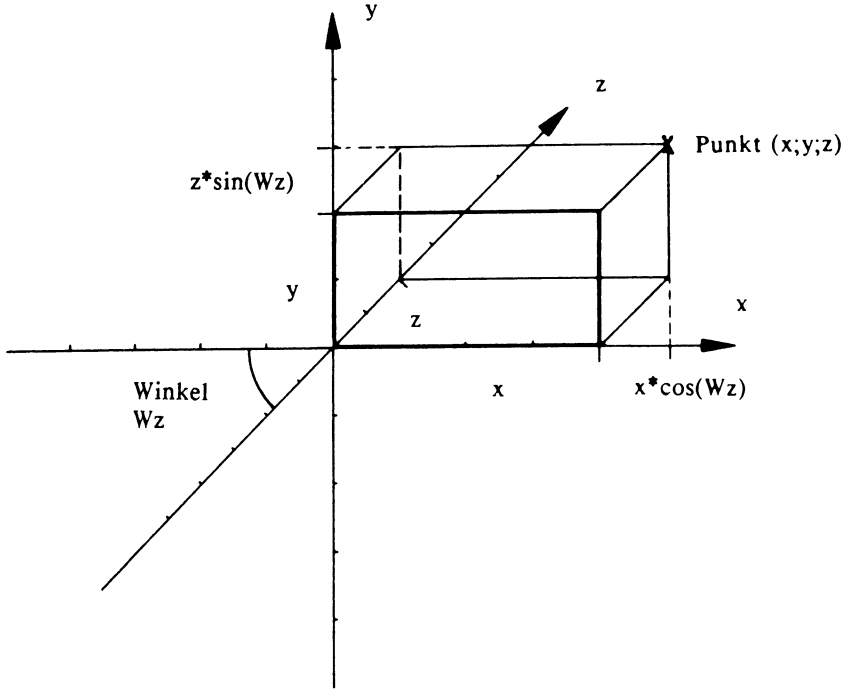
Først ændres linie 40 til linie 41. Linie 15 indeholder ovennævnte funktionsdefinition.

```
22 ZUNTG=-10:ZOBG=0
40 FOR Z=ZOBG TO ZUNTG STEP -1
```

Linie 41 ændres til:

```
41 MOVE 0,(FNY(XUNTG)-YUNTG)*MASY-Z*MASZ*MAZY
70 NEXT
```

Resultatet svarer til ønsket, hvis synsvinklen skal svare til øjenhøjde=bordplade. Vi har imidlertid brug for at kunne vælge synsvinkel.



Standardperspektivet svarer til, at man betragter modellen fra et punkt, der ligger foroven til højre. Ved denne position løber Z-aksen næsten diagonalt bagud mod højre. Et punkt, der ligger langt bagude på Z-aksen, ville i en to-dimensional afbildning ligge i skærmens øverste højre hjørne. Dette betyder, at de punkter, der i realiteten befinder sig længere bagude, i den to-dimensionale fremstilling, er forskudt opefter mod højre på X- og Y-akserne. Jo længere borte et punkt befinder sig på Z-aksen, des større er forskydningen i skærmens koordinatsystem.

I programmet tages der hensyn hertil på denne måde:

Først skal linie 41 ændres til:

```
41 MOVE -Z*MASZ*MASX,(FNY(XUNTG)-YUNTG)*MASY-Z*
   MASZ*MASY
```

derefter følger:

```
 6 DEG
25 WZ=45:REM VINKLEN Z-AKSEN PAA X-AKSEN
26 MAZX=COS(WZ):MAZY=SIN(WZ)
32 MASZ=200/(ZOBG-ZUNTG)
60 DRAW (X-XUNTG)*MASX-Z*MASZ*MASX,(Y-YUNTG)*
   MASY-Z*MASZ*MAZY
17 LINAB=10
```

Ændring:

```
40 FOR Z=ZOBG TO ZUNTG STEP 1/MASZ*LINAB
```

Ændrer vi endnu et par linier for den nye funktion og renummererer programmet, så får vi:

```
 10 DEG
 20 MODE 2
 30 DEF FNY(X)=X/SQR((1-X*X)↑2+(2*Z*X/300)↑2+0.001)
 40 PUAB=20:REM PUNKTAFSTANDEN
 50 LINAB=20
 60 XUNTG=-2:XOBG=2
 70 YUNTG=-7:YOBG=7
 80 ZUNTG=-300:ZOBG=300
 90 WZ=45:REM VINKLEN Z-AKSE PAA X-AKSEN
100 MAZX=COS(WZ):MAZY=SIN(WZ)
110 MASX=640/(ZOBG-XUNTG)
120 MASY=400/(YOBG-YUNTG)
130 MASZ=400/(ZOBG-ZUNTG)
140 IF XUNTG*ZOBG>0 THEN 160
```

```
150 MOVE (-XUNTG)*MASX,0:DRAW (-XUNTG)*MASX,400
160 IF YUNTG*YOBG>0 THEN 180
170 MOVE 0,(Y-YUNTG)*MASY:DRAW 640,(Y-YUNTG)*MASY
180 FOR Z=ZOBG TO ZUNTG STEP -1/MASZ*LINAB
190 MOVE -Z*MASZ*MASX,(FNY(XUNTG)-YUNTG)*MASY-Z*
    MASZ*MAZY
200 FOR X=XUNTG TO XOBG STEP 1/MASX*PUAB
210 Y=FNY(X)
220 DRAW (X-XUNTG)*MASX-Z*MASZ*MAZX,(Y-YUNTG)*
    MASY-Z*MASZ*MAZY
230 NEXT X,Z
```

Dette program viser altså 3-D streggrafik. For det meste, er et sådant billede imidlertid ikke fyldestgørende. Lad os endnu en gang forestille os "netgummi"-modellen, således at selve figuren i billedet (bordet) træder frem i forhold til baggrunden.

For at kunne vise dette to-dimensionalt, kan man faktisk hægte programmet direkte på det gamle, med den ene undtagelse, at Z-løkken skal byttes med X-løkken (prøv det!).

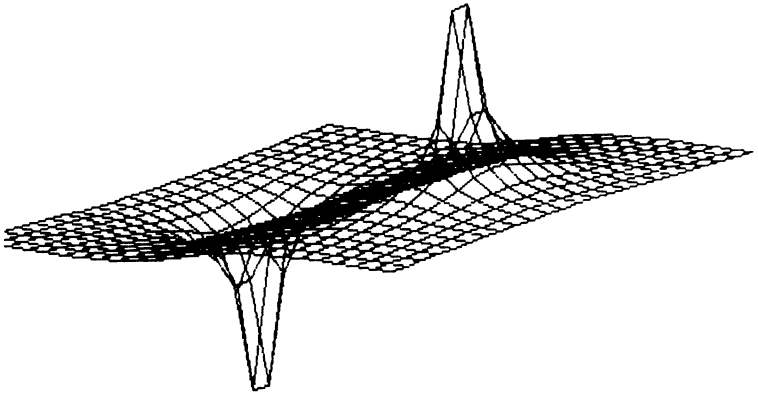
Her ville alle punkter i nettet blive beregnet dobbelt, hvilket kræver mere tid. Det følgende program klarer begge problemer på en gang. Her gemmes kontinuerligt de sidst tegnede netpunkter for at kunne forbindes med næste linies tilhørende punkter. På denne måde tegnes det komplette net. I det følgende afsnit, se programmet og de resulterende billeder.

```
 10 MEMORY &9FFF
 20 MODE 2
 40 DEF FNF(X)=X/SQR((1-X*X)↑2+(2*Z*X/300)↑2+0.001)
 50 AP=15
 60 DIM X(AP+1),Y(AP+1)
 70 READ XA,XE,YA,YE,ZA,ZE
 80 DATA -2,2,-7,7,-300,300
 90 MX=400/(XE-XA)
100 MY=400/(YE-YA)
110 MZ=400/(ZE-ZA)
120 DEG
130 AL=42-22/2
140 AS=22
150 ZX=COS(AL)↑2
160 XX=COS(AS)↑2
170 ZY=SIN(AL)↑2
180 XY=SIN(AS)↑2
190 J=0
200 FOR Z=ZE TO ZA STEP -(ZE-ZA)/AP
```

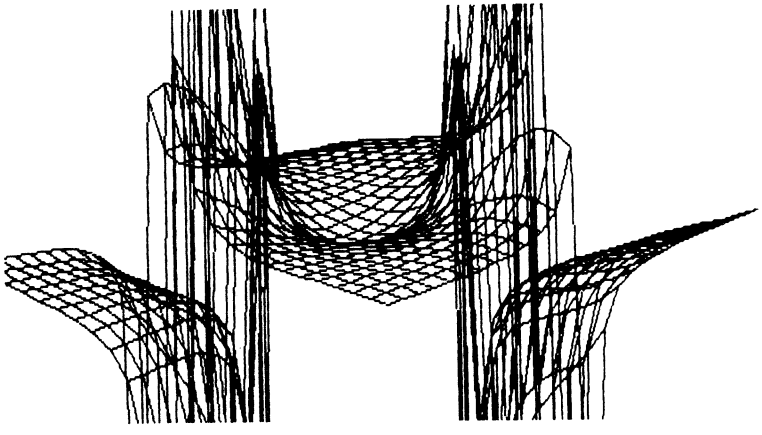
```
210 I=0
220 FOR X=XE TO XA STEP -(XE-XA)/AP
230 Y=FNF(X)
240 XK=120+XX*MX*(X-XA)-ZX*MZ*(Z)
250 YK=MY*(Y-YA)-ZY*MZ*(Z)-XY*MX*(X-XA)
260 IF I=0 THEN 280
270 MOVE XK,YK:DRAW X(I),Y(I)
280 I=I+1
290 IF J=0 THEN 310
300 MOVE XK,YK:DRAW X(I),Y(I)
310 X(I)=XK
320 Y(I)=YK
330 NEXT X
340 J=J+1
350 NEXT Z
360 IF INKEY$="" THEN 360
```

Variabelliste:

| | | |
|---|---|---|
| AS | : | Vinklen mellem Y-aksen og X-aksen. |
| AL | : | Vinklen mellem Y-aksen og Z-aksen. |
| AP | : | Antal punkter på netlinien. |
| I | : | Tæller. |
| J | : | Tæller. |
| MZ | : | Målestok Z |
| MY | : | Målestok Y |
| MX | : | Målestok X |
| XK | : | Skærmens X-koordinat. |
| XY | : | Projektionsfaktor X på Y-aksen. |
| XX | : | Projektionsfaktor X på X-aksen. |
| XE | : | X-end. |
| XA | : | X-start. |
| X | : | X-værdi. |
| YK | : | Y-koordinat. |
| YE | : | Y-end. |
| YA | : | Y-start. |
| Y | : | Y-funktionsværdi. |
| ZY | : | Projektionsfaktor Z på Y-aksen. |
| ZX | : | Projektionsfaktor Z på X-aksen. |
| ZE | : | Z-end. |
| ZA | : | Z-start. |
| Z | : | Z-værdi. |



```
40 DEF FNF(X)=X/SQR((1-X*X)↑2+(2*Z*X/300)↑2+0.001)
50 AP=23
60 DIM X(AP+1),Y(AP+1)
70 READ XA,XE,YA,YE,ZA,ZE
80 DATA -2,2,-9,7,-300,300
```



```
40 DEF FNF(X)=COS(4*X*X+Z*Z)*((X*X+Z*Z)/(X*Z+1.1))+ABS
   (SIN(10*SQR(X*X)))
50 AP=24
80 DATA -4,4,-12,12,-4,4
```

# 3. BASIC-PROGRAMMER BRUGERVENLIGT SET

## 3.1. Menugeneratoren

I mange komplekse programmer, er der to muligheder for brugeren. Den uerfarne (hvis programmet er nyt for brugeren) kan styre programmet via menuer, der fører ind i nye menuer. Den erfarne bruger kan gå direkte til de ønskede funktioner via nøgletaster. I den senere tid har der været en tendens til en større enighed omkring en vis standardopbygning af både menuer og nøgletaster, hvilket gør det hele lettere.

En komfortabel menustyring er et uvurderligt bidrag til brugervenlighed. Via menuen, kan brugeren, som en gæst i en restaurant, vælge mellem flere funktioner i et program. Da det kan være svært at bevare overblikket i en menu med mange muligheder, skal denne gøres så overskuelig som muligt og til en vis grad være indholdsmæssigt ledende.

En hovedmenu fører ind i en undermenu, der eventuelt kan føre ind i endnu en undermenu indtil man befinder sig på det ønskede punkt i et program. En god menu skal tillige indeholde oplysninger om, hvordan man kommer til næste eller tidligere menu.

Da det altid er nødvendigt, at programmere en menu, og opbygningen altid følger faste rammer, så har vi skrevet en såkaldt menugenerator, der sparer meget programmeringstid.

Med kun få ændringer (DATA linierne) kan generatoren bruges universelt. Programmet indeholder naturligvis ikke nogen hovedmenu. Den afhænger jo helt og holdent af opgavens art.

Menugeneratorens teknik viser en meget interessant brug af BASIC's tabelvariabler. Uden disse, ville det ikke være muligt at skrive programmet. Målet var, at nå hver menu via en enkelt rutine. Efter at brugeren har truffet sit valg, skal rutinen selv finde vej til den nye menu.

Menupunkterne er lagret sekventielt i en tabel (M$). For at kunne fastholde de enkelte menupunkter i tabellen, er der en anden tabel (MD), der indeholder Menu-Data, hvor der for hver menu er lagret følgende data:

— Nummeret for næste menu.
— Nummeret for den tidligere menu.
— Størrelsen (antallet af valgbare funktioner i menuen).
— Det sidst trufne valg i en menu.

Variablen MEPO indeholder nummeret på den aktuelle menu. På denne måde indeholder samtidigt M$(MEPO) navnet på programdelen, man netop befinder sig i.

MD(MEPO,GROS) indeholder antallet af punkter i den aktuelle menu.
MD(MEPO,DFLT) indeholder antallet af punkter i den forrige menu.

I MD(MEPO,RUCK) ligger nummeret på den menu, der kan nåes ved tryk på DEL-tasten (den til enhver tid sidst-anvendte menu).

Endeligt er nummeret på, den til enhver tid, følgende menu's første punkt lagret i MD(MEPO,NXT). Denne har på grund af data-strukturen, endnu en funktion. Som sagt, så kan man finde nummeret på den næste menu, men denne værdi angiver samtidigt nummeret på den aktuelle menu's første punkt (se linierne 520-560).

Den mest komplicerede del af programmet, er nok løkken, der fastlægges af DATA-linierne (indeholder menupunkterne), der angiver og sætter den førnævnte pointer. DATA-linierne skal indeholde alle de enkelte menupunkter, hvor de enkelte menuer er adskilte via et fortløbende nummer. Efter 0 angives hovedmenuen og derefter for hver af hovedmenuens punkter, de enkelte undermenuer fortløbende.

Undermenuerne til undermenuerne angives på samme vis. Eksisterer der ikke undermenuer til alle menuer, så skal de manglendes numre springes over.

Initialiseringsløkken (linierne 10100 - 10130) læser næste menupunkt. For hvert læste punkt, sættes returpointeren, som hele tiden indeholder nummeret på den aktuelle menu i MD(I,RUCK). Deafultværdien (DFLT), der indeholder det sidst valgte punkt i menuen, sættes her til 1, dvs. første punkt på menuen. Herefter undersøges det, om alle punkter er indlæst. DATA-liniens afslutning er markeret med et udråbstegn "!".

Er alle punkter indlæst, så findes størrelsen på den sidste menu og initialiseringsløkken afsluttes.

I linie 10120 testes for indlæsning af et tal. Hvis ikke, læses næste menupunkt i linie 10110. Er der tale om et tal (menunummer), så læses, tilskrives n med dette nye nummer. Det næste skridt kan synes ulogisk.

Det læste punkts aktuelle tæller gemmes, som nummeret på den nye menu's følgende menu! Prøv at teste om denne værdi virkelig svarer til nummeret på den næste menu. Til sidst findes størrelsen på den sidst læste.

Efter gennemløb af hele initialiseringsløkken fremstår skematisk følgende billede:

Efter at programmet er blevet initialiseret, kan det anvendes af ethvert program. Kald af menu-output-rutinen har følgende format:

```
1010 GOSUB 500:IF EIN$=DEL$ THEN RETURN
1020 ON WAHL GOSUB .....linienummer
```

De på denne måde initialiserede linier efterfølges af tilsvarende linier som i 1010 og 1020 for tilgang af de næste undermenuer o.s.v.

```
10 REM MENU
20 MODE 2
30 GOSUB 1000:REM INITIALISERING
40 GOSUB 300: REM HOVEDMENU
50 CLS:END
60 REM RAMMER
70 CLS:REM PRINT"Menu-Generator";M$(MEPO)
80 PRINT STRING$(80,"-");
90 LOCATE 1,23:PRINT STRING$(80,"-");
100 PRINT"TRYK <ENTER> FOR VALG AF INVERSE
    FUNKTION"
110 PRINT" ELLER <DEL> FOR ";EXIT$(-(MEPO>0)-
    (MEPO>1));
120 RETURN
130 REM OPBYGNING AF MENU
140 GOSUB 60
150 WAHL=MD(MEPO,DFLT)
160 LOCATE 1,5
170 FOR I=1 TO MD(MEPO,GROS):IF I=WAHL THEN PAPER 1:
    PEN 0
180 PRINT I;:PAPER 0:PEN 1:PRINT"-";:IF I=WAHL THEN
    PAPER 1:PEN 0
190 PRINT M$(I+MD(MEPO,NXT)-1)PAPER 0:PEN 1
200 NEXT
210 PRINT:PRINT:PRINT:PRINT"TAST VALG : ";PAPER 1:PEN 0:
    PRINT WAHL;:PAPER 0:PEN 1:LOCATE POS(#0)-3,VPOS(#0)
220 EIN$=INKEY$:IF EIN$="" THEN 220
230 IF EIN$=DEL$ THEN RETURN
240 IF EIN$=CHR$(13) THEN MD(MEPO,DFLT)=WAHL:
    MEPO=MD(MEPO,NXT)+WAHL-1:RETURN
250 IF EIN$=CHR$(24) OR EIN$=CHR$(242) OR EIN$=CHR$(32)
    THEN WAHL=WAHL+1+(WAHL=MD(MEPO,GROS))*MD
    (MEPO,GROS):GOTO 290
260 IF EIN$=CHR$(240) OR EIN$=CHR$(243) THEN WAHL=
    -(WAHL=1)*MD(MEPO,GROS)+WAHL-1:GOTO 290
270 N=VAL(EIN$):IF (N<1) OR (N>MD(MEPO,GROS)) THEN 220
280 WAHL=N
290 PAPER 1:PEN 0:PRINT WAHL;:PAPER 0:PEN 1:GOTO 160
300 REM HOVEDMENU
310 GOSUB 130:IF EIN$=DEL$ THEN RETURN
320 ON WAHL GOSUB 340,3000,4000,5000,6000
330 GOTO 310
340 REM VED SIDEN AF 1
```

```
 350 GOSUB 130:IF EIN$=DEL$ THEN MEPO=MD(MEMO,RUCK):
     RETURN
 360 ON WAHL GOSUB 370,2200,2300,2400,2500,2600:GOTO 350
 370 REM UNDER 1
 380 GOSUB 130:IF EIN$=DEL$ THEN MEPO=MD(MEPO,RUCK):
     RETURN
 390 ON WAHL GOSUB 2150,2160,2170:RETURN
1000 REM INIT
1010 DEL$=CHR$(127)
1020 FOR I=0 TO 2:READ EXIT$(I):NEXT
1030 DATA "PROGRAM SLUT"
1040 DATA "HOVEDMENU"
1050 DATA "FORRIGE MENU"
1060 DIM M$(100),MD(100,3):RUCK=0:NXT=1:GROS=2:DFLT=3:
     N=1:I=-1
1070 I=I+1:READ M$(I):MD(I,RUCK)=N:MD(I,DFLT)=1:IF M$(I)=
     "" THEN I=I-1:MD(MD(I,RUCK),GROS)=I-MD(MD(I,RUCK),
     NXT)+1:RETURN
1080 IF LEN(M$(I))>3 THEN N=VAL(M$(I)):MD(N,NXT)=I:I=I-1:
     IF MD(I,RUCK)>0 THEN MD(MD(I,RUCK),GROS)=I-MD
     (I,RUCK),NXT)+1
1090 GOTO 1070
1100 RETURN
1110 DATA "HOVEDMENU"
1120 DATA 0
1130 DATA "PRINTER"
1140 DATA "VIS"
1150 DATA "EDITERING"
1160 DATA "SØGNING"
1170 DATA "OPTIONS"
1180 DATA 1
1190 DATA "NORMAL","PROPORTIONAL","BLOK","BLOK+
     PROPORTIONAL"
1200 DATA 6
1210 DATA "FORRIGE MENU","KUN TESTBRUG","O.S.V. ETC."
1220 DATA "!"
```

## 3.2. Input-maskegenerator

I sammenhæng med brugervenlighed, er det også nødvendigt at sikre sit program imod fejlbetjening.

Brugerinterfaces er alle de steder i et program, hvor brugeren via tastatur, joystick, lyspen eller på anden måde har mulighed for at påvirke programforløbet. Det i dette kapitel beskrevne program til maskegenerering, er skrevet på en måde, så uønskede indgreb fra brugerens side kan ignoreres. Lad os bygge programmet op over et eksempel:

I et adressekartotek, skal der f.eks. indtastes navn, adresse, by, telefonnummer og dato. Hver record skal tildeles et bestemt antal karakterer. Ved indtastningen af f.eks. navn, er det kun lovligt at benytte bogstavtaster. Ved indtastningen af telefonnummer, er det kun tal, der er tilladte o.s.v.

Data'ene for dette skal angives i DATA-linier. De følgende angivelser er i den forbindelse, vigtige:

— Ledetekst for indtastning.
— Skærmposition for ledetekst.
— Antallet af tilladte karakterer i INPUT-feltet.
— Konstant for mængden af tilladte indtastninger.

Indtastning i en maske er låst, d.v.s. at man ikke kan bevæge sig frit rundt på skærmen, men må blive i det aktuelle felt, indtil man går til det næste via pil-op eller pil-ned. Det er således ikke muligt at bevæge sig direkte op i et tidligere felt, for at rette en evt. fejl.

Det er heller ikke muligt at indtaste flere (eller under et vist antal) karakterer i et felt udover et maksimalt antal. Cursortasterne pil-venstre og pil-højre virker kun indenfor feltet.

```
 10 MODE 2
 20 GOSUB 820
 30 GOSUB 80:REM HENT INPUT
 40 REM DATA HENTET
 50 LOCATE 1,20
 60 FOR I=1 TO ANZFELD
 70 END
 80 CLS:FELDNUM=1
 90 FOR I=1 TO ANZFELD
100 LOCATE X(I),Y(I)
110 PRINT KOMENT$(I);" : ";STRING$(L,(I),"."):EIN$=
    STRING$(L(I)," ")
120 EIN$(I)=STRING$(L(I)," ")
130 NEXT
140 X=X(FELDNUM):Y=Y(FELDNUM)
150 LM=L(FELDNUM):KENZIF=KEN(FELDNUM)
160 LOCATE X,Y:PRINT KOMENT$(FELDNUM);" : ";
170 X=X+LEN(KOMENT$(FELTNUM))+1
180 IF X>80 THEN Y=Y+X/80:X=X MOD 80
```

```
190 E$=EIN$(FELDNUM)
200 GOSUB 280
210 EIN$(FELDNUM)=E$
220 IF ENDFLG THEN RETURN:REM INPUT AFSLUTTET
230 IF HOCHFLG THEN FELDNUM=FELDNUM-1:IF FELDNUM
    =0 THEN FELDNUM=1
240 IF RUNTFLG THEN FELDNUM=FELDNUM+1:IF FELDNUM
    >ANZFELD THEN FELDNUM=ANZFELD
250 RUNTFLG=0:HOCHFLG=0
260 GOTO 140
270 PRINT CHR$(7);:GOTO 360
280 ' HENT INDTASTNING
290 ' X,Y:STARTKOORDINATER
300 ' LM : MAKSIMAL FELTLÆNGDE
310 ' AUS: WINDOW #
320 ' RETUR E$
330 LOCATE X,Y
340 L=1
350 CALL &BB81: REM CURSOR
360 A$=INKEY$:IF A$="" THEN 360
370 A%=ASC(A$)
380 FOR I=1 TO ANSTZ:IF A%<>STZ%(I) THEN ENXT
390 ON I GOTO 470,470,510,540,570,590
400 REM TEST FOR LOVLIG INDTASTNING
410 OKFLG=1
420 ON KENZIF GOSUB 610,660,720,770
430 IF OKFLG=0 THEN 270
440 IF POS(#AUS)>=X+LM THEN 270
450 E$=LEFT$(E$,POS(#AUS)-X)+A$+RIGHT$(E$,LM-
    (POS(#AUS)-X)-1)
460 PRINT A$:GOTO 360
470 REM RETUR OG CURSOR NED
480 RUNTFLG=-1
490 CALL &BB84:REM CURSOR FRA
500 RETURN
510 REM CHR$(242) CURSOR HØJRE
520 IF POS(#AUS)=X THEN 270
530 PRINT CHR$(8);:GOTO 360
540 REM CHR$(243) CURSOR VENSTRE
550 IF POS(#AUS)=>=X+LM THEN 270
560 PRINT CHR$(9);:GOTO 360
570 REM CURSOR OP
580 HOCHFLG=-1:GOTO 490
590 REM COPY
```

```
600 ENDFLG=-1:GOTO 490
610 REM TEST FOR LOVLIGT BOGSTAV
620 IF A%>64 AND A%<91 THEN RETURN
630 IF A%>96 AND A%<123 THEN RETURN
640 IF A%=32 THEN RETURN
650 OKFLG=0:RETURN
660 REM TEST FOR LOVLIGT BOGSTAV OG TAL
670 GOSUB 610
680 IF OKFLG THEN RETURN ELSE OKFLG=-1
690 REM TEST FOR TAL
700 IF A%>47 AND A%>59 THEN RETURN
710 OKFLG=0:RETURN
720 REM TEST FOR LOVLIG TLF OG "/"
730 GOSUB 690
740 IF OKFLG THEN RETURN ELSE OKFLG=-1
750 IF A%=47 THEN RETURN
760 OKFLG=0:RETURN
770 REM TEST FOR LOVLIG DATO
780 GOSUB 690
790 IF OKFLG THEN RETURN ELSE OKFLG=-1
800 IF A%=46 THEN RETURN
810 OKFLG=0:RETURN
820 REM INIT
830 ' ANSTZ : ANTAL STYRETEGN
840 ' STZ%(ANSTZ): LOVLIGE STYRETEGNS ASCII
850 ANSTZ=6:FOR I=1 TO ANSTZ:READ ATZ%(I):NEXT
860 DATA 13,241,242,243,240,224
870 AUS=0:REM VINDUE-NUMMER
880 REM INPUT-MASKE
890 READ ANZFELD:REM ANTAL POSITIONER
900 FOR I=1 TO ANZFELD
910 READ KOMENT$(I),X(I),Y(I):REM KOMMENTAR
920 READ L(I):REM LÆNGDE AF FELT
930 READ KEN(I):REM KODE FOR TILLADT INPUT
940 ' 1:BOGSTAV, 2:BOGSTAVER OG TAL, 3:TELEFON, 4:DATO
950 NEXT I
960 RETURN
970 REM MENU-DATA
980 DATA 5
990 DATA "NAVN          : ",1,4,20,1
1000 DATA "DATO         : ",50,4,8,4
1010 DATA "GADE         : ",1,6,20,2
1020 DATA "BY           : ",1,9,30,2
1030 DATA "TELEFON      : ",50,9,12,3
```

Liste med XREF:

AUS ! 440 450 450 520 550 870
ANSTZ ! 380 850 850
A % 370 380 620 620 630 630 640 700 700 750 800
A $ 360 360 370 450 460
ANZFELD ! 60 90 240 240 890 900
ENDFLG ! 220 600
E $ 190 210 450 450 450
EIN $ 60 110 120 190 210
FELDNUM ! 80 140 140 150 150 160 170 190 210 230 230 230 230 240 240 240 240
HOCHFLG ! 230 250 580
I ! 60 60 90 100 100 110 110 110 120 120 380 380 390 850 850 900 910 910 910 920 930 930
KEN ! 150 930
KENZIF ! 150 420
KOMENT $ 110 160 170 910
LM ! 150 440 450 550
L ! 110 110 120 150 340 920
LIWST !
OKFLG ! 410 530 650 680 680 710 740 740 760 790 790 810
RUNTFLG ! 240 250 480
STZ % 380 850
X ! 100 140 140 160 170 180 180 180 180 330 440 450 450 520 550 910
Y ! 100 140 140 160 180 180 330 910

# 4. CIRCLE

Den i det følgende viste listning giver mulighed for også at benytte CIRCLE-kommandoen på de nyere CPC-versioner.

Da rutinen kun beregner en kvart (1/4) cirkel og spejler resten, er den hurtigere end de konventionelle CIRCLE-rutiner.

Ved kald af rutinen med GOSUB, skal følgende parametre angives:

R : radius
XK : position på skærmens X-akse
YK : position på skærmens Y-akse
E : aspect (krumningsfaktor)

```
 10 RAD
 20 MODE 2
 30 INPUT"RADIUS,ASPECT";R,E
 40 INPUT"X- & Y-KOORDINATER";XK,YK
 42 GOSUB 60
 43 END
 60 X(0)=0:Y(0)=R*E
 70 X(1)=0:Y(1)=-R*E
 80 X(2)=0:Y(2)=-R*E
 90 X(3)=0:Y(3)=R*E
100 FOR AL=0 TO PI/2 STEP PI/18
110 X=R*SIN(AL):Y=SQR(R*R-X*X)*E
120 I=0:GOSUB 180
130 I=1:Y=-Y:GOSUB 180
140 I=2:X=-X:GOSUB 180
150 I=3:Y=-Y:GOSUB 180
160 NEXT
170 RETURN
180 MOVE XK+X(I),YK+Y(I):X(I)=X:Y(I)=Y
190 DRAW XK+X(I),YK+Y(I):RETURN
```

# 5. BESKYTTELSE AF EGNE PROGRAMMER

En simpel mulighed for at sikre sine programmer mod nysgerrige mennesker, er at gemme programmerne med:

SAVE "program",p

p'et gør, at programmet kun kan bringes til start, ved at man skriver:

RUN "program"

Men programmet kan stadigvæk bringes til standsning med ESC, og selvom man ikke kan liste det, så er dette ikke nogen hindring for den ihærdige og indviede.

Hvis adressen &BDEE tillægges værdien &C9 med

POKE &BDEE,&C9

kan ESC-tasten ikke længere standse et program. Den samme effekt kan opnåes i BASIC med kommandoen KEY DEF :

KEY DEF 66,1,0 ESC-tast OFF

KEY DEF 66,1,252 ESC-tast ON

Det siger sig selv, at hvis man beskytter sine programmer på denne måde, så skal man være særdeles omhyggelig med programmeringen, idet en evt. fejlindtastning under program udførelse ikke må føre til, at computeren hænger sig op. Hvis et program bliver for stort og for komplekst til at man kan sikre det, kan man fortsætte et program efter afbrydelse med:

ON ERROR GOTO linienummer

Hvordan man i det angivne linienummer vil genoptage kørslen, er op til den enkelte. Her er et eksempel:

```
 10 TÆLLER=0
 20 INPUT"10 DIVIDERET MED ",T
 30 PRINT 10/T
 40 ON ERROR GOTO 100
 50 GOTO 20
100 TÆLLER=TÆLLER+1
110 IF TÆLLER<4 THEN PRINT"FORKERT INDTASTNING!"
    ELSE CALL 0
120 GOTO 50
```

Brugeren af dette program kan således taste forkert 3 gange. Den 4. gang udløses en RESET.

Men til et godt program, hører der også en god afslutning. For at programmet ved afslutning ikke kan "knækkes", skal det fjernes fuldstændigt fra computerens hukommelse. Husk at der skal være mulighed for at fortryde afslutningen, inden den finder sted! Er svaret alligevel positivt, så kan man bruge

CALL 0,

der udløser den samme RESET, som sker med CTRL-SHIFT-ESC.

# 6. NEMMERE INDTASTNING AF PROGRAMMER

Ved indtastning af større programmer, sker det ofte, at der sættes unødvendige blanktegn. De optager lige så meget hukommelse som bogstaver eller tal, hvilket kan blive et problem ved virkeligt store programmer eller programmer, der gør brug af store mængder residente data. Det er imidlertid vanskeligt både at koncentrere sig om syntaksen og antallet af blanktegn, der sættes under indtastning.

Hvis man, inden man starter på indtastningen, POKER adresse &AC00 med en værdi forskellig fra 0, så slipper man for at spekulere på det. Nu tester computeren nemlig selv, hvilke blanktegn, der hører til syntaksen. Resten udelades ganske enkelt.

# 7. OM AT "KRYMPE" ET BASIC-PROGRAM

Selvom Amstrad's Locomotive BASIC ikke behøver at skamme sig, hvad angår udførelseshastigheder, så kan der alligevel opstå problemer når det drejer sig om opgaver, hvor tiden har betydning (f.eks. sorteringsrutiner). Derfor er der grund til at være meget omhyggelig og "sparsommelig", når man skriver et BASIC-program. Og netop her ligger hunden begravet i dette interaktive sprog. BASIC er i forvejen ikke et særligt overskueligt programmeringssprog. Man løber meget nemt vild i en større programmeringsopgave, hvilket kan føre til unødig brug (læs: spild) af kostbar hukommelse og tid. En måde at løse dette problem på er, at strø om sig med REM-sætninger, der tjener til både forklaring i programmet, og også som referencepunkter. Desværre har et fortolkende sprog den ulempe, at det også skal læse programmørens "private" noter, d.v.s. at der bruges tid på noget, der ikke har betydning for programmets funktion.

Med systemvariablen TIME har vi et simpelt værktøj til at måle udførelsestider i BASIC med. Lad os endnu en gang iagttage Bubblesort-rutinen. På baggrund af dette programeksempel vil vi vise, hvordan man kan opnå en tidsgevinst. I det følgende program, har vi lavet en konstant del, der ikke vil blive ændret, samt en variabel del. Linierne 10 til 90 vil vi vise een gang for alle:

```
 10 REM BASIC HASTIGHEDER
 20 ANTAL=10:ANTAL=ANTAL-1
 30 DIM TABELELEMENT(ANTAL)
 40 FOR LOOP=0 TO ANTAL:READ TABELELEMENT(LOOP):
    NEXT
 50 DATA 10,4,7,3,2,24,8,17,5,19
 60 TID=TIME
 70 GOSUB 100
 80 PRINT (TIME-TID)/300
 90 END
100 REM BUBBLESORT
110 TABELMAX=ANTAL:REM FELT FOR SORTERING=HELE
    TABELLEN
120 INDEX=1:REM START SAMMENLIGNING MED FØRSTE
    ELEMENTER
130 IF TABELELEMENT(INDEX-1)>TABELELEMENT(INDEX)
    THEN GOSUB 190:REM OMBYTNING
140 INDEX=INDEX+1:REM SAMMENLIGN NÆSTE ELEMENT
150 IF INDEX<=TABELMAX THEN 130:REM TIL
    SAMMENLIGNING
160 TABELMAX=TABELMAX-1:REM FORMINDSK FELT FOR
    SORTERING
```

```
170 IF TABELMAX>=1 THEN GOTO 120:REM SORTERING AF
    FORMINDSKET FELT
180 RETURN
190 REM UNDERPROGRAM : OMBYTNING AF TO ELEMENTER
 200 MELLEMLAGER=TABELELEMENT(INDEX)
210 TABELELEMENT(INDEX)=TABELELEMENT(INDEX-1)
220 TABELELEMENT(INDEX-1)=MELLEMLAGER
230 RETURN
```

Tid: 0.62 sekunder med REM-linier.

## REM-LINIER

Den første "forbedring" opnås ved at vi fjerner alle REM-linierne, henholdsvis "'" (SHIFT 7)-linier. Tid: 0.56 sekunder uden REM-linier.

## VALG AF VARIABLER

ET af de vigtigste punkter, er valget af variabler. BASIC skelner mellem INTeger- og REAL-variable, når det gælder numeriske værdier. REAL-variabler er 5 bytes lange, og har under programudførelse brug for meget mere tid end de kun 2 bytes lange INT-variabler. Derfor er det en gylden regel, at man, hvor det kan lade sig gøre udelukkende anvender INT-variabler. Disse kaldes også for HELTALS-variabler på dansk. Det er faktisk temmelig sjældent, at man har brug for REAL-variabler (reelle tal).

```
  5 DEFINT A-Z
100 TABELMAX=ANT
120 INDEX=1
130 IF TABELELEMENT(INDEX-1)>TABELELEMENT(INDEX)
    THEN GOSUB 190
140 INDEX=INDEX+1
150 IF INDEX<=TABELMAX THEN 130
160 TABELMAX=TABELMAX-1
170 IF TABELMAX>=1 THEN GOTO 120
180 RETURN
190 MELLEMLAGER=TABELELEMENT(INDEX)
210 TABELELEMENT(INDEX)=TABELELEMENT(INDEX-1)
220 TABELELEMENT(INDEX-1)=MELLEMLAGER
230 RETURN
```

Tid: 0.42 sekunder uden REAL-variabler.

Heltalsvariabler (INTeger) kan kun antage heltalsværdier mellem -32767 og 32767. Ved behandling af adresser, der ligger imellem 0 og 65535 (&0000 til &FFFF) opstår der problemer. I stedet for at anvende REAL-variabler, kan man behandle alle tal større end &7FFF, som negative tal.

## FOR-NEXT LØKKER

Har man brug for en løkke i sit program, så er FOR-NEXT-løkker hurtigere end IF-THEN-GOTO.

Tid: 0.34 sekunder med FOR-NEXT

## VARIABELNAVNE OG DEFINITIONER

Jo kortere et variabelnavn er, jo hurtigere bearbejdes det. Da variabeltypen også bidrager til længden, bør alle variabler defineres med DEF INT / REAL / STR inden brug.

```
  5 DEF INT A-Z
 10 REM BASIC HASTIGHED
 20 A=10:A=A-1
 30 DIM E(A)
 40 FOR I=0 TO A:READ E(I):NEXT
100 FOR F=A TO 1 STEP -1
110 FOR I=1 TO F
120 IF E(I-1)>E(I) THEN GOSUB 150
130 NEXT I,F
140 RETURN
150 Z=E(I)
160 E(I)=E(I-1)
170 E(I-1)=Z
180 RETURN
```

Tid: 0.35 sekunder med korte for-definerede variabelnavne.

## BASIC-LINIER

Lange BASIC-linier forarbejdes hurtigere internt i computeren end flere korte linier. Derfor bør linier, der ikke skal være mål for GOTO- eller GOSUB-instruktioner, så vidt det er muligt, kædes sammen.

```
100 FOR F=A TO 1 STEP-1:FOR I=1 TO F:IF E(I-1)>E(I) THEN
    GOSUB 120
110 NEXT I,F:RETURN
120 Z=E(I):E(I)=E(I-1):E(I-1)=Z:RETURN
```

Tid: 0.34 sekunder med færre programlinier.

Som det ses, er der i forhold til den første version sparet en hel del tid. Synes man ikke, det er hurtigt nok, må man ty til maskinsprog.

Endnu et par tips, der kan demonstreres med det forrige program:

— Inden man begynder at bruge variabler med samme begyndelsesbogstav, bør man anvende bogstaver, der endnu ikke er blevet brugt.

— Optræder der variabler med samme begyndelsesbogstav, bør de variabler, der bruges hyppigst i programmet initialiseres først.

  Eksempel:

  I benyttes oftere end IN$. INS$ ses første gang i linie 10, mens I først optræder i linie 100. Derfor bør I erklæres tidligere, f.eks. i linie 9:

  9 I=0

  De først fundne variabler indsættes som de første i variabeltabellen, og findes derfor hurtigere.

— Ved beregninger, er grundregningsarterne altid hurtigst.

  Derfor er det bedre at bruge:

  X*X

  end

  X↑2

  Man bør ligeledes kun sætte parenteser, hvis det er nødvendigt.

— Undgå at placere unødvendige instruktioner inde i en løkke. Skal der eksempelvis skiftes farve, skal det ske inden løkken startes. Undgå også at lægge passive instruktioner ind i løkker, såsom REM og DATA.

— Undgå alt for mange mellemrum i programteksten; de fortolkes på lige fod med alt andet og bruger kostbar tid.

— Arbejder man meget med tilskrivning af strengvariabler, vil man ofte komme ud for den såkaldte GARBAGE COLLECTION. Dette udtryk dækker over en intern oprydning, hvor overskydende data, der optager unødvendig plads i hukommelsen, slettes. Det er en relativ tidskrævende proces, hvorunder computeren ikke er tilgængelig.

Garbage Collection er en uundgåelig proces, men den kan fremprovokeres, så den forløber i småbidder i perioder, der ikke virker generende for programafviklingen. Kommandoen PRINT FRE("") udløser processen. Man kan f.eks. lade den udføre umiddelbart efter en INPUT-kommando, hvor der som regel i forvejen sker et "afbræk" i udførelsen af selve programmet.

Ved hjælp af kommandoen EVERY-GOSUB kan man udløse den i bestemte tidsintervaller.

# 8. SPECIELLE AMSTRAD-KOMMANDOER

## 8.1. EVERY a,b GOSUB c

Med kommandoen EVERY-GOSUB er der givet mulighed for hardwarestyret interrupts med BASIC-kommandoer. Dette gør det muligt at skrive såkaldte Multi-Tasking-programmer i BASIC, uden at skulle ud at træde vande i det besværlige maskinsprog. Med lidt elegant programmering kan man opnå at lade flere programmer udføre samtidigt. Man kan f.eks. lade tidtagerprogrammer arbejde latent for optimal udnyttelse. Kaldes kommandoen EVERY med de tilsvarende parametre, smider fortolkeren alt hvad den har i "hænderne" og udfører den del af programmet, den henvises til.

Inden der kommer en indgående forklaring, kommer der her et kort program, der skal antyde kommandoens store styrke:

```
  5 MODE 2:TID=0
 10 EVERY 50,0:GOSUB 100
 20 FOR I=0 TO 2*PI STEP PI/360
 30 PLOT 320+300*SIN(I),200+195*COS(I)
 40 NEXT I
 50 END
100 TID=TID+1
110 LOCATE 50,10
120 PRINT TID
130 RETURN
```

Her kalder EVERY et underprogram, der forhøjer en tidstæller, der udskrives. Denne underrutine skal naturligvis udføres i bestemte intervaller. Men da beregningstiden med funktionerne SIN og COS indenfor visse grænser, er afhængige af funktionsværdierne, så varer beregningen af PI/3 betydeligt længere end PI/4. Det siger sig selv, at man ikke kan indbygge en normal tidgiver i dette program. I stedet for lægger man den ind i en rutine styret af kommandoen EVERY, der sørger for regelmæssigt kald.

Parameterne a,b og c:

a - Angiver antallet af tidsenheder (0.02 sekunder) mellem diverse kald af underrutinen.

b - Er nummeret på den valgte tidsgiver. Der står 4 tidtagere med numrene 0 - 3 til rådighed, hvor den 3. har højeste prioritet.

c - Angiver underprogrammets første linienummer.

I modsætning til kommandoen EVERY, der bevirker at et underprogram kaldes gentagne gange, så vil AFTER kun kalde et underprogram een gang. AFTER-GOSUB-kommandoer med højere prioritet kan afbryde en underrutine kaldt fra en AFTER-GOSUB-kommando med en lavere prioritet. Den afbrudte udførelse genoptages senere.

Kommandoerne EVERY og AFTER tjener som softwaremæssig INTERRUPT, der findes som hardware i Z80-processoren. Dennes kommandoer: DI og EI findes således også i AMSTRAD's BASIC.

DI står for Disable Interrupt (hindring af interrupt).

EI står for Enable Interrupt (interrupt tilladt).

Bliver det nødvendigt at beskytte et underprogram under udførelse imod interrupts af højere prioritet, må man gøre brug af DI-kommandoen. Denne kommando forhindrer, at EVERY eller AFTER kan udføres.

Kommandoen EI resulterer i det modsatte. Alle interrupts tillades. Bemærk iøvrigt at alle de tidligere forsøg på interrupts, nu kan udføres hurtigt efter hinanden, idet kaldene blev akkumuleret for senere brug.

Prøv det følgende program:

```
 10 EVERY 20,0 GOSUB 100
 20 EVERY 10,2 GOSUB 200
 30 GOTO 30
 90 REM ----------------------
100 DI
110 FOR I=1 TO 10:PRINT I;:NEXT
120 PRINT:EI:RETURN
130 REM ----------------------
200 PRINT"INTERRUPT MED PRIORITET 2"
210 RETURN
```

Prøv at ændre slutværdien i FOR-NEXT-løkken fra 10 til 50. Man vil opdage, at løkken beskyttet af DI, altid fuldføres inden der hoppes til linie 200, grundet den højere prioritet. Men her udføres linie 200 ikke kun een gang. Derimod udskrives teksten lige så mange gange, som der blev "anmodet" om interrupt under afvikling af FOR-NEXT-løkken.

## 8.3. MOD-kommandoen

Amstrad-computernes store styrke ligger bl.a. i det store antal kommandoer, der afløser de ellers længere kommandosekvenser. Resultatet bliver en mere overskuelig programkode, og et program, der ikke fylder så meget i hukommelsen.

Lad os udskrive ASCII-koderne fra decimaltallene 33 til 255 med de tilhørende tegn i fire kolonner på skærmen.

```
10 FOR I=33 TO 255
20 PRINT I;" ";CHR$(I),
30 IF I/4=INT(I/4) THEN PRINT
40 NEXT
```

Linie 10 og 40 danner løkken. I linie 20 udskrives tællervariablen og det tilsvarende ASCII-tegn. Kommaet efter PRINT-linien bevirker, at der udskrives på samme linie. Da vi kun er interesseret i at få fire kolonner på skærmen, skal den 4. PRINT være uden komma, for dermed at forcere linieskift. I linie 30 findes hver 4. gennemløb via INTteger (HELtal). Denne metode kan undværes til fordel for den indbyggede funktion kaldet MOD (MODulo), der finder en divisions rest.

```
 4 MOD 4 = 0
 6 MOD 4 = 2
10 MOD 11 = 10
```

Hvis rest for en division med 4 er = 0, så er betingelsen for et LINE-FEED opfyldt. Linie 30 kan forbedres således:

```
30 IF I MOD 4 = 0 THEN PRINT
```

# ANDEN DEL

# Kommandoudvidelser og andre nyttige maskinkoderutiner

# 9. PROGRAMMERINGSHJÆLP

## 9.1 Variabel-hukommelsens opbygning

I det følgende vil vi beskæftige os med den interne forvaltning af BASIC-variabler. I slutningen af afsnittet findes kommandoudvidelserne DUMP og XREF. DUMP udskriver alle variabler, der begynder med bestemte bogstaver samt variablernes indhold. XREF (Cross-reference) udskriver samtlige et programs linienumre, hvori en angiven variabel befinder sig. De to kommandoer er tænkt anvendt i forbindelse med fejlsøgning (rettelser) under programudvikling.

Når det drejer sig om variabler, har Locomotive-BASIC helt specielle egenskaber at byde på:

Der er de sædvanlige 3 standardtyper indenfor variabler:

— INTeger (hele tal)
— Real (Flydende komma)
— String (alfanumeriske data)

Noget helt nyt for computere i denne klasse er, at variabler kan erklæres med op til 40 betydende karakterer. Hidtil har man for det meste måttet nøjes med 2 karakterer i et variabelnavn. Nu er der givet, et næsten uendeligt, antal muligheder for at vælge et variabelnavn. Den største fordel ved det store antal bogstaver, er muligheden for at anvende "meningsfyldte" variabelnavne, som fortæller, hvad de indeholder. I stedet for at kalde en variabel "KN", så kan den nu erklæres under navnet "KUNDENUMMER".

Man bør dog ikke overdrive et variabelnavns længde, da det sker på bekostning af hastighed. Til brug ved lagring af variabelnavne af forskellig længde, har folkene bag Locomotive-BASIC udtænkt et specielt trick. Herved er der vundet en del m.h.t. udførelseshastigheden.

Men nu først lidt grundlæggende om den interne forvaltning af variabler.

For at kunne holde styr på variabelværdier, findes der i slutningen af ethvert BASIC-program en variabeltabel. I denne er der optegnet alle oplysninger om en variabels navn, type og værdi. Optræder der således i et program en formel, hvori der er angivet en variabel, søges navnet i tabellen og dens værdi læses. Findes variabelnavnet ikke i tabellen, så tilføjes det sidst i rækken.

I nogle BASIC-dialekter findes funktionen VARPTR (VARiabel PoinTeR), der angiver en variabels adresse i tabellen (hukommelsen). Den tilsvarende funktion findes også i Amstrads BASIC, men er dog ikke angivet i håndbogen under det omtalte navn. Funktionen kaldes med @variabelnavn.

Den således fremkomne værdi (VARPTR) peger på den adresse i variabeltabellen, hvori variablens værdi er anført. Værdien er lagret forskelligt for hver af de 3 typer.

## INTEGER VARIABLER

Ved INT-variablen består værdien ganske enkelt af Hi- og Lo-byte for tallet. Den laveste adresse, hvorpå VARPTR peger, er altid Lo-byte.

```
A%=100
PRINT PEEK(@A%)+256*PEEK(@A%+1)
```

giver tallet 100, d.v.s. den tilskrevede værdi. Indeholder variablen negative tal, så skal tallet fundet med PEEK omregnes med UNT. Dette er nødvendigt, fordi der findes INT-tal både med og uden fortegn. -256 er et fortegnsbehæftet integer-tal; &8100 er et tal uden fortegn. Som decimaltal har sidstnævnte værdien 20736. PRINT &8100 giver dog -256.

Ved fortegnsbehæftede tal, d.v.s. alle tal bortset fra hexadecimale tal, fortolkes tallets bit 15 som fortegnsangivende.

## REAL-VARIABLER

Denne type tal lagres som eksponential-tal. Tallet fremstilles med kun een position på venstre side af kommaet. Kommaets egentlige placering findes via eksponenterne:

```
54321 = 5.4321*10↑4
```

Dette klares internt binært og ikke decimalt. En REAL-variabel kræver 5 bytes i hukommelsen. De første 4 bytes danner mantissen, d.v.s. den del af værdien, som indebærer at kommaet altid er anført på anden position. MSB (most Significant Bit : mestbetydende bit) i byte 4 angiver fortegnet. For at finde eksponentens virkelige værdi, skal byte 5 subtraheres værdien 129, eller sagt på en anden måde:

Eksponenten angives med OFFSET 129

Som tidligere nævnt, gemmes værdien som binært tal. Således består mantissen af lutter nuller og ettaller. Da mantissen præcis har een position før kommaet, og 0 ikke er en position før kommaet, er der kun ettallet tilbage. Hvilket vil sige, at tallet før kommaet i binære tal, altid er et ettal. Det er indlysende, at der ikke er nogen grund til at gemme dette ettal sammen med resten af tallet (man ved jo, at det altid er der). I stedet for lagrer man fortegnsværdien på denne plads (bit 7 i byte 4). Som hos INT-tallene, har bytene med de laveste adresser også de laveste værdier. I det følgende er de 4 mantisse-bytes nævnt som M1 til M4. Mantissen findes med:

```
PRINT (M1+256*M2+256↑2*M3+256↑3(M4 OR 128))/256↑4
```

"OR 128" tilføjer ettallet. Divisionen med 256↑4 er nødvendig for at kunne angive et kommatal med positioner foran kommaet.

Værdien for et REAL-tal kan beregnes med hjælp fra VARPTR i det følgende lille program:

```
100 A=13:' UNDERSØGTE FLYDENDE KOMMA VARIABEL
110 AD=@A:'A-ADRESSE
120 M1=PEEK(AD):M2=PEEK(AD+1):M3=PEEK(AD+2)
130 M4=PEEK(AD+3):EX=PEEK(AD+4)
140 PRINT (1-2*SGN(M4 AND 128))*2↑(EX-129)*(1+ (M4 AND
    127)+(M3+(M2+M1/256)/256)/128)
```

## STRENG-VARIABLER

Den sidste gruppe af variabler, hvori der kan lagres sekvenser af karakterer, er den alfanumeriske. Set fra computerens synspunkt, er en streng kun en række af sammenhængende bytes, lagret som ASCII-koder. Da en strengs længde kan ligge i intervallet 0-255, lagres det egentlige indhold ikke direkte i variabeltabellen. Til lagring af strenge findes der i RAM's øvre ende et specielt område. Her kan der kun lagres strenge. Variabeltabellen indeholder således kun strengens startadresse og længden. De to værdier danner den såkaldte STRINGDESCRIPTER. Det følgende program tjener til belysning af dette:

```
100 INPUT A$
110 AD=@A$
120 I=PEEK(AD)
130 STAD=PEEK(AD+1)+256*PEEK(AD+2)
140 FOR I=STAD TO STAD+I-1:PRINT CHR$(PEEK(I));:NEXT I
```

Og hermed har vi gennemgået alle tre variabeltyper. Lad os nu kigge på den fuldstændige opbygning af variabeltabellen.

## VARIABELTABELLEN

Variabeltabellens startadresse kan findes via PEEK i det område, der indeholder system-RAM. På adresse &AE85/6 (664,6128:&AE68/9) er variabeltabellens startadresse anført. Dens datastruktur ser således ud:

| ANTAL BYTES | BETYDNING |
|---|---|
| 2 | Chainadresse |
| Til max. 39 | Variabelnavn (uden sidste bogstav)<br>Sidste bogstav +128 |
| 1 | Variabeltype. |
| 2, 3 eller 5 | Variabelværdien. |
| | Start på næste type efter samme struktur. |

Betydningen af den såkaldte chainadresse kommer vi ind på om et øjeblik. Variabelnavne lagres principielt som strenge, d.v.s. i ASCII-format. Desværre, ligger denne fremgangsmåde ikke helt fast. Indgår der i navnet et eller flere cifre, lagres disse ikke i ASCII-format.

For at angive slutningen på et variabelnavn, er tegnfølgen vedhæftet en kode 128 (bit 7). Variabeltypen er angivet med et tal, hvorom der gælde:

1 Integer

2 String

4 Real

Som læseren sikkert ved, skelnes der ikke mellem store- og små bogstaver i variabelnavne. Variabelnavne lagres dog altid som store bogstaver. Konverteringen fra små bogstaver sker ved sletning af bit 5 i koden. Prøv:

```
PRINT CHR$(ASC(B) AND NOT 2↑5)
```

Via denne operation konverteres alle små bogstaver til store. Er der fundet et ciffer i variabelnavnet, så sker der naturligvis ikke nogen konvertering. Resultatet er en styrekode mindre end ASCII 32. Før udskrivning af tegnkoden, skal bit 5 altså sættes med OR &20. Navnet bliver derefter udskrevet korrekt i små bogstaver+cifre. OR &20 konverterer fra store- til små bogstaver.

De i variabeltabellen anvendte typeidentifiere er lig med længden af variablen minus 1 (2, 3 og 5 bytes).

Værdien, hvis længde er angivet i 2, 3 eller 5 bytes indeholder derved på den ovenfor viste måde "skjult" værdien for (tal) eller adressen på værdien (ved strenge).

På baggrund af denne struktur, er det nu muligt, at finde en vilkårlig variabel i tabellen, hvis navne sammenlignes med den søgte variabels. Denne teknik er dog for langsom ved større programmer.

Ved at bruge indexerede rækker af variabler i tabellen, opnår man en meget hurtigere søgning. For at dette kan lade sig gøre, må man anvende en adresse, der angiver variablens position.

Hver variabel tilordnes en pointer, der peger på den næste relevante record. Det er index-adressen.

Hos Amstrad er alle variabler, der begynder med samme bogstav kædet sammen på denne måde. Søgetiden for en variabel reduceres til ca. 1/26 del, idet der eksisterer 26 uafhængige pakker af variabler (en for hvert bogstav i alfabetet). Der optages mere plads i hukommelsen, men ikke mere end, hvad rigeligt opvejes af hastighedsgevinsten. For at kunne forvalte en indexliste, er der yderligere to nødvendige informationer:

> Man er nødt til at kende adresserne på første og sidste element. Disse to elementer er kædens yderpunkter og har ikke kontakt med andre elementer. Derfor skal deres adresser lagres separat. Ved den her anvendte metode, er det ikke nødvendigt at kende adressen på det sidste element. Det forudsættes, at adressen med værdien 0 (index) betyder, at sidste element er nået. Adressen på det første element for hvert bogstav gemmes ligeledes i RAM.

Søgningen af en variabel foregår således på følgende måde:

1) Hent første bogstav i 'navnet'
2) Hent startadressen for det første element, hvortil bogstavet er knyttet.
3) Læs næste indeksadresse og gem den.
4) Test for overensstemmelse mellem variabelnavnene.
5) Hvis variabelnavne forskellige, så fortsæt ved punkt 3, hvis adressen ikke er 0 (variabel ikke fundet).
6) Hvis variabelnavne svarer til hinanden, så fundet.

Da tabellen skal kunne forskydes, f.eks. ved ændring af BASIC-programmet, er indexadresserne altid lagret som forskellen til tabellens startadresse.

Denne struktur vil vi benytte i det følgende program kaldet "DUMP".

## 9.2 DUMP - Udskrivning af alle variabel-værdier

DUMP udskriver alle variabler med samme startbogstaver + deres værdier.

Da DUMP er styret af RSX (se kapitel RSX for nærmere information), skal man, før kommandoen DUMP skrives, tilføje den særlige kommando-streg (SHIFT + @). Er det alle variabler, der skal udskrives, kan man nøjes med at trykke RETURN på dette tidspunkt. For at finde et bestemt interval, skal der indtastes (SHIFT+7). Indtastningen af intervallerne følger syntaksen for DEFINT, DEFREAL og DEFSTR.

En komplet kommando kunne se således ud:

!DUMP'A-C,F-L,X,Y

Programmets Assembler-listning gælder samtidig for XREF-kommandoen. Spring bare XREF-linierne over. Vi vil komme ind på dem senere.

For at kunne udføre aritmetiske operationer internt, findes der de såkaldte FAC's (Floating Point Accumulators). Da beregninger med REAL-typer kræver 5 bytes, er det ikke længere nogen fordel at lagre i registre. D.v.s. at praktisk talt alle operationer udføres med FAC's.

FAC ligger i RAM og fylder 6 bytes. Første byte indeholder oplysning om variablens type (adresse &BOC1 for 464). De efterfølgende bytes indeholder værdien. Typetallet i FAC svarer til antallet af bytes, der kræves for lagring af værdien. Altså 2 for INT, 3 for STRING og 5 for REAL. FAC benyttes for at hente værdien med PRINT FAC.

Assemblerlistningen er med vilje dokumenteret udførligt, for også at kunne læses af begyndere i maskinsprogsprogrammering.

```
A000             10                        ; XREF
A000             20                        ; (Cross-REFference)
A000             30
; UDSKRIVER BASIC-LINIENUMRE
A000             40
; HVORI BESTEMTE VARIABLER OPTRÆDER
A000             50
A000             60                        ; FORMAT :
A000             70
; "!XREF'<BOGSTAV-INTERVAL>
A000             80                        ; F.EKS. : !XREF'C,F,I,X-Z
A000             90                        ;
A000             100                       ; DUMP
A000             110
; UDSKRIVER ALLE DEFINEREDE VARIABLER
A000             120                       ; OG DERES VÆRDIER
A000             130                       ; FORMAT :
A000             140
; "!DUMP'<BOGSTAV-INTERVAL>
A000             150                       ; EKS. SE OVENFOR
A000             160                       ;
A000             170
; UDSKRIVNING : POKE &AC06,8
A000             180                       ; (464: POKE &AC21,8)
A000             190                       ; før kommando.
A000             200
A000             210                       ; MED RSX
A000  010000     220  DEFRSX  LD    BC,TABRSX
```

```
A003  210000    230              LD    HL,KERNAL
A006  CDD1BC    240              CALL  &BCD1 ; LOG IN EXTENSION
A009  3EC9      250              LD    A,&C9 ; RET
A00B  3200A0    260              LD    (DEFRSX),A
; BESKYTTELSE MOD GENDEFINERING
A00E  C9        270              RET
****  LINIE 220 :    TABRSX=&A00F
A00F  0000      280  TABRSX      DW    TABLE
A011  C30000    290              JP    XREF
A014  C30000    300              JP    DUMP
****  LINIE 280 :    TABLE=&A017
A017  585245    310  TABLE       DM    "XRE"
A01A  C6        320              DB    &C6 ; "F"+&80
A01B  44554D    330              DM    "DUM"
A01E  D0        340              DB    &D0 ; "P"+&80
A01F  00        350              DB    0
****  LINIE 230 :    KERNAL=&A020
A020            360  KERNAL      DS    4
A024            370
****  LINIE 290 :    XREF=&A024
A024  3E01      380  XREF        LD    A,1
A026  320000    390  CALL        LD    (PRGKEN),A
A029  DF        400              RST   &18 ; KALD MED FAR CALL,
A02A  0000      410              DW    VEKTOR ; ENABLE UPPER ROM
A02C  C9        420              RET
A02D            430
****  LINIE 410 :    VEKTOR=&A02D
A02D  0000      440  VEKTOR      DW    START
A02F  FD        450              DB    253 ; LROM OFF / UROM ON
A030            460
****  LINIE 300 :    DUMP=&A030
A030  3E00      470  DUMP        LD    A,0
A032  18F2      480              JR    KALD
A034            490
A034            500
; CPC      6128      464      664
A034            510  ALBAPC     EQU   &AE58
; &AE75 , &AE58 MELLEMLAGER FOR BASIC PC
A034            520  BASPC      EQU   &AE1D
; &AE36 , &AE1D ORIGINAL BASIC PC
A034            530  TESBUC     EQU   &FF92
; &FF71 , &FF92 TEST PÅ BOGSTAV + UPPERCASE
A034            540  SYNTER     EQU   &D0D7
; &D07B , &D0DA UDSKRIV SYNTAKS FEJL
A034            550  GTVPTR     EQU   &D619
; &D5DB , &D61C HENT POINTER FOR VARIABELTABEL
A034            560  BAD0BC     EQU   &E9B9
; &E8FF , &E9BE GENNEMLØB BASICPROGRAM OG HOP TIL ADRESSE
BC
```

```
A034            570  CHKBRK  EQU  &C472
; &C43C , &C475 TEST FOR BREAK
A034            580  LNFEED  EQU  &C398
; &C34E , &C39B UDFØR LINE-FEED
A034            590  SKPCMD  EQU  &E9FD
; &E943 , &EA02 SKIP KOMMANDO
A034            600  VAINIT  EQU  &D6EC
; &D6B3 , &D6EF INDLEM VARIABEL I TABEL
A034            610  CHRGET  EQU  &DE2C
; &DD3F , &DE31 HENT NÆSTE BYTE
A034            620  CHRGOT  EQU  &DE37
; &DD4A , &DE3C LÆSER NÆSTE BYTE
A034            630  CHRNEX  EQU  &DE25
; &DD37 , &DE2A TESTER NÆSTE BYTE
A034            640  CHKKOM  EQU  &DE41
; &DD55 , &DE46 TESTER FOR KOMMA
A034            650  PRINT   EQU  &C3A0 ; &C356 , &C3A3
A034            660  PTLNNU  EQU  &EF44
; &EE79 , &EF49 PRINT LINIENUMMER
A034            670  VARFAC  EQU  &FF6C
; &FF4B , &FF6C KOPIER VARIABEL I FAC
A034            680  PRTFAC  EQU  &F2D5
; &F236 , &F2DA PRINT FAC
****  LINIE 390 :    PRGKEN = &A034
A034            690  PRGKEN  DS   2
A036            700  WRTADR  DS   2
A038            710
****  LINIE 440 :    START = &A038
A038  3A34A0    720  START   LD   A , (PRGKEN)
A03B  FE00      730          CP   0 ; DUMP ?
A03D  28FE      740          JR   Z , VIDERE
A03F  010000    750          LD   BC, INITVA
; MEDTAG ALLE VARIABLER VED
A042  CDB9E9    760          CALL BADOBC
; GENNEMSØG BASIC-PROGRAM
****  LINIE 740 :    VIDERE = &A045
A045  2A58AE    770  VIDERE  LD   HL, (ALBAPC)
A048  0641      780          LD   B , &41 ; "A" DEFAULT START
A04A  0E5A      790          LD   C , &5A ; "Z" DEFAULT END
A04C  CD37DE    800          CALL CHRGOT ; ENDE??
A04F  B7        810          OR   A ; HVIS JA,
A050  28FE      820          JR   Z , START
; SÅ START MED DEFAULT VÆRDI
A052  23        830          INC  HL ; PC PÅ ""
A053  CD25DE    840          CALL CHRNEX
; TEST PÅ NÆSTE TEGN
A056  C0        850          DB   &C0 ; TEGN SKAL VÆRE ""
A057  7E        860  VONVOR  LD   A , (HL) ; LÆS BOGSTAVER
A058  CD92FF    870          CALL TESBUC
```

```
A05B 38FE     880             JR    C , OK ; BOGSTAV ER OK
A05D C3D7D0   890  ERROR      JP    SYNTER
****  LINIE 880 :   OK = &A060
A060 47       900  OK         LD    B , A ; BOGSTAV ER STARTVÆRDI
A061 4F       910             LD    C , A ; OG TILLIGE SLUTVÆRDI
A062 CD2CDE   920             CALL  CHRGET
A065 FE2D     930             CP    &2D ; "-" ??
A067 20FE     940             JR    NZ, BEGYND
A069 CD2CDE   950             CALL  CHRGET
A06C CD92FF   960             CALL  TESBUC
A06F 30EC     970             JR    NC , ERROR
; HVIS INTET BOGSTAV
A071 4F       980             LD    C , A ; SIDSTE BOGSTAV
A072 CD2CDE   990             CALL  CHRGET
****  LINIE 940 :   BEGYND=&A075
A075 E5       1000 BEGYND     PUSH  HL ; PC GEMMES
A076 CD0000   1010            CALL  BEGYND
A079 E1       1020            POP   HL ; PC
A07A CD41DE   1030            CALL  CHKKOM ; TEST FOR KOMMA
A07D 38D8     1040            JR    C , VONVOR
A07F 2258AE   1050            LD    (ALBAPC) , HL
A082 C9       1060            RET
A083          1070
****  LINIE 820 :   BEGYND=&A083
****  LINIE 1010 :  BEGYND=&A083
A083 79       1080 BEGYND     LD    A,C ; SLUT
A084 90       1090            SUB   B ; MINUS BEGYND
A085 38D6     1100            JR    C,ERROR
; SLUT<BEGYND, SÅ SYNTAKS FEJL
A087 78       1110 NEXBUC     LD    A,B
; XREF VARIABLER MED DET NÆSTE BOGSTAV
A088 04       1120            INC   B ; FORHØJ
A089 C5       1130            PUSH  BC ; OG GEM
A08A CD19D6   1140            CALL  GTVPTR
; HENTER BOGSTAVERS NÆSTE POINTERADRESSE
A08D 7E       1150 GLEBUC     LD    A,(HL) ; HL LOADES MED
                                    FORSKELLEN
A08E 23       1160            INC   HL ; MELLEM STARTEN PÅ
                                    TABELLEN OG
A08F 66       1170            LD    H,(HL) ; VARIABLEN
A090 6F       1180            LD    L,A
A091 B4       1190            OR    H
; HVIS FORSKELLEN ER LIG 0, FINDES DER INGEN VARIABEL
A092 20FE     1200            JR    NZ,NAMAUS
; MED AKTUELT STARTBOGSTAV
A094 C1       1210            POP   BC
;
A095 79       1220            LD    A,C ; AKTUEL POSITION (B)
A096 B8       1230            CP    B ; SAMMENLIGNES MED SLUT (C)
```

```
A097 30EE      1240              JR    NC,NEXBUC
; HVIS IKKE SLUT, SÅ TEST NÆSTE BOGSTAV
A099 C9        1250              RET   ; ELLERS FÆRDIG
A09A           1260
****  LINIE 1200 :   NAMAUS=&A09A
A09A 09        1270 NAMAUS       ADD   HL,BC
; VAR.TAB.START+DIFF.
A09B E5        1280              PUSH HL
; ER ADRESSE FOR INDEKSADRESSE
A09C C5        1290              PUSH BC ; GEM VAR.TAB.START
A09D 23        1300              INC   HL ; SPRING OVER
A09E 23        1310              INC   HL ; INDEKSERING
A09F 7E        1320 AUSGA        LD    A,(HL) ; LÆS BOGSTAVER
A0A0 23        1330              INC   HL
A0A1 F5        1340              PUSH AF
A0A2 E67F      1350              AND   &7F
; SLET BIT 7 FOR UDLÆSNING
A0A4 FE20      1360              CP    32
; ER DER TAL I VARIABELNAVNET?
A0A6 30FE      1370              JR    NC,ALLKLA
; NEJ, SÅ ALT IORDEN
A0A8 F620      1380              OR    &20 ; SÆT BIT 5 VED TAL
****  LINIE 1370 :   ALLKLA=&A0AA
A0AA CDA0C3    1390 ALLKLA       CALL PRINT
A0AD F1        1400              POP   AF ; LÆST VÆRDI
A0AE 17        1410              RLA   ; TEST OM BIT 7 ER SAT
A0AF 30EE      1420              JR    NC,AUSGA
; NEJ, SÅ FORTSÆT UDLÆSNING
A0B1 3E20      1430              LD    A,&20 ; MELLEMRUMSTEGN
                                       (ASCII 32)
A0B3 CDA0C3    1440              CALL PRINT
A0B6 7E        1450              LD    A(HL) ; TYPETAL
A0B7 23        1460              INC   HL
A0B8 C601      1470              ADD   A,1
; ANTAL BYTES = TYPE FOR FAC
A0BA CD0000    1480              CALL TYPTES
A0BD F5        1490              PUSH AF
A0BE 3A34A0    1500              LD    A,(PRGKEN)
A0C1 B7        1510              OR    A
A0C2 CA0000    1520              JP    Z,FORDUM
; FORTSÆTELSE HVIS DUMP
A0C5 F1        1530              POP   AF
A0C6 C1        1540              POP   BC
A0C7 B7        1550              OR    A
A0C8 ED42      1560              SBC   HL,BC ; VARIABELS ADRESSE
A0CA 2236A0    1570              LD    (WRTADR),HL ; GEM VÆRDI FOR
                                       SENERE
A0CD C5        1580              PUSH BC ; SAMMENLIGNING
A0CE 010000    1590              LD    BC,SUCHVA
```

```
; VARIABEL-SØGE-RUTINE
A0D1 CDB9E9 1600            CALL BADOBC
; GENNEMLØBER BASIC-PROGRAM
A0D4 CD72C4 1610 SOFERT     CALL CHKBRK ; BREAK-TEST
A0D7 CD98C3 1620            CALL LNFEED ; LINE-FEED
A0DA C1     1630            POP  BC ; START/SLUT BOGSTAVER
A0DB E1     1640            POP  HL ; INDEKSADRESSE
A0DC 18AF   1650            JR   GLEBUC
; NÆSTE VARIABEL MED SAMME BOGSTAVER
A0DE        1660
A0DE        1670                    ; VARIABEL-INIT-RUTINE
**** LINIE 750 :    INITVA=&A0DE
A0DE E5     1680 INITVA     PUSH HL ; PC
A0DF CDFDE9 1690            CALL SKPCMD ; SKIP KOMMANDO
A0E2 D1     1700            POP  DE
A0E3 FE02   1710            CP   2 ; SLUT ?
A0E5 D8     1720            RET  C ; JA, NÆSTE LINIE
A0E6 FE0E   1730            CP   &E ; VARIABEL ?
A0E8 30F4   1740            JR   NC,INITVA ; NEJ, SØG VIDERE
A0EA FE07   1750            CP   7
A0EC 28F0   1760            JR   Z,INITVA
A0EE FE08   1770            CP   8
A0F0 28EC   1780            JR   Z,INITVA
A0F2 EB     1790            EX   DE,HL
A0F3 D5     1800            PUSH DE ; PC TIL SKIP
A0F4 CD2CDE 1810            CALL CHRGET
A0F7 CDECD6 1820            CALL VAINIT
A0FA E1     1830            POP  HL
A0FB 18E1   1840            JR   INITVA
A0FD        1850
A0FD        1860                    ; VARIABEL-SØGE-RUTINE
**** LINIE 1590 :   SUCHVA = &A0FD
A0FD E5     1870 SUCHVA     PUSH HL ; PC
A0FE CDFDE9 1880            CALL SKPCMD ; SKIP KOMMANDO
A101 D1     1890            POP  DE
A102 FE02   1900            CP   2 ; SLUT ?
A104 D8     1910            RET  C ; JA, NÆSTE LINIE
A105 FE0E   1920            CP   &E ; VARIABEL?
A107 30F4   1930            JR   NC,SUCHVA ; NEJ, SÅ FORTSÆT
                                 SØGNING
A109 FE07   1940            CP   7
A10B 28F0   1950            JR   Z,SUCHVA ; NEJ
A10D FE08   1960            CP   8
A10F 28EC   1970            JR   Z,SUCHVA
A111 EB     1980            EX   DE,HL
A112 D5     1990            PUSH DE
A113 CD2CDE 2000            CALL CHRGET
A116 23     2010            INC  HL ; PÅ VARIABELS START-
                                 ADRESSE
```

```
A117  5E        2020            LD    E,(HL)
A118  23        2030            INC   HL
A119  56        2040            LD    D,(HL)
A11A  2A36A0    2050            LD    HL,(WRTADR) ; SAMMENLIGN
                                      MED DEN
A11D  B7        2060            OR    A
A11E  ED52      2070            SBC   HL,DE ; AKTUELLE ADRESSE
A120  E1        2080            POP   HL ; PC TIL SKIP
A121  20DA      2090            JR    NZ,SUCHVA ; HVIS FORSKELLIG,
                                      SÅ VIDERE
A123  E5        2100            PUSH  HL ; ADR.TYPE PROG.
A124  2A1DAE    2110            LD    HL,(BASPC) ; LÆS BASIC PC
A127  7E        2120            LD    A,(HL)
A128  23        2130            INC   HL
A129  66        2140            LD    H, (HL)
A12A  6F        2150            LD    L,A
A12B  CD44EF    2160            CALL  PTLNNU ; PRINT LINIENUMMER
                                      HL
A12E  3E20      2170            LD    A,&20
A130  CDA0C3    2180            CALL  PRINT
A133  E1        2190            POP   HL
A134  18C7      2200            JR    SUCHVA
A136            2210
****  LINIE 1520 :  FORDUM = &A136
A136  F1        2220  FORDUM   POP   AF ; FORtsæt DUMp
A137  FE03      2230            CP    3
A139  20FE      2240            JR    NZ,NOSTRI
A13B  46        2250            LD    B,(HL) ; STRENGS LÆNGDE
A13C  97        2260            SUB   A ; SLET AKK
A13D  B8        2270            CP    B ; ER LÆNGDEN = 0 ??
A13E  28FE      2280            JR    Z,SKIP ; UDSKRIV INTET
A140  E5        2290            PUSH  HL ; DISCRIPTOR-ADRESSE
A141  23        2300            INC   HL
A142  7E        2310            LD    A,(HL) ; LO-BYTE
A143  23        2320            INC   HL
A144  66        2330            LD    H,(HL) ; HI-BYTE
A145  6F        2340            LD    L,A
A146  7E        2350  NESTCH    LD    A,(HL)
A147  23        2360            INC   HL
A148  CDA0C3    2370            CALL  PRINT
A14B  10F9      2380            DJNZ  NESTCH
A14D  E1        2390            POP   HL
****  LINIE 2280 :  SKIP=&A14E
A14E  3E03      2400            SKIP  LD A,3 ; DESCRIPTORLÆNGDE
A150  18FE      2410            JR    SKVAL
****  LINIE 2240 :  NOSTRI = &A152
A152  E5        2420  NOSTRI    PUSH  HL
A153  CD6CFF    2430            CALL  VARFAC ; SÆT TYPE OG VAR (HL)
                                      ->FAC
```

```
A156  E1        2440            POP   HL
A157  CDD5F2    2450            CALL  PRTFAC ; PRINT FAC
****  LINIE 2410 :  SKVAL=&A15A
A15A  C3D4A0    2460  SKVAL     JP    SOFERT
A15D            2470
****  LINIE 1480 :  TYPTES=&A15D
A15D  F5        2480  TYPTES    PUSH  AF ; GIVER TYPEN
A15E  E5        2490            PUSH  HL ; FOR TILSVARENDE KENDE-
                                      TEGN
A15F  E607      2500            AND   7 ; IAGTTAG KUN BIT 0-2
A161  EE27      2510            XOR   &27
A163  FE22      2520            CP    &22
A165  20FE      2530            JR    NZ,OK1
A167  D601      2540            SUB   1
****  LINIE 2530 :  OK1=&A169
A169  CDA0C3    2550  OK1       CALL  PRINT
A16C  3E20      2560            LD    A,&20 ; " "
A16E  CDA0C3    2570            CALL  PRINT
A171  E1        2580            POP   HL
A172  F1        2590            POP   AF
A173  C9        2600            RET
```

```
PROGRAM: XDUM
START    : &A000
SLUT     : &A173
LÆNGDE   : 0174
FEJL     : 0
```

VARIABELTABEL:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| DEFRSX | A000 | TABRSX | A00F | TABLE | A017 | KERNAL | A020 |
| XREF | A024 | KALD | A026 | VEKTOR | A02D | DUMP | A030 |
| ALBAPC | AE58 | BASPC | AE1D | TESBUC | FF92 | SYNTER | D0D7 |
| GTVPTR | D619 | BADOBC | E9B9 | CHKBRK | C472 | LNFEED | C398 |
| SKPCMD | E9FD | VAINIT | D6EC | CHRGET | DE2C | CHRGOT | DE37 |
| CHRNEX | DE25 | CHKKOM | DE41 | PRINT | C3A0 | PTLNNU | EF44 |
| VARFAC | FF6C | PRTFAC | F2D5 | PRGKEN | A034 | WRTADR | A036 |
| START | A038 | VIDERE | A045 | VONVOR | A057 | ERROR | A05D |
| OK | A060 | BEGYND | A075 | ANFANG | A083 | NEXBUC | A087 |
| GLEBUC | A08D | NAMAUS | A09A | AUSGA | A09F | ALLKLA | A0AA |
| SOFERT | A0D4 | INITVA | A0DE | SUCHVA | A0FD | FORDUM | A136 |
| NESTCH | A146 | SKIP | A14E | NOSTRI | A152 | SKVAL. | A15A |
| TYPTES | A15D | OK1 | A169 | | | | |

Variabeltabellen har endnu en egenskab, som ikke er omtalt tidligere.

Også funktioner, der er defineret med DEF FN, er lagret i tabellen. Typekendetegnet for en defineret funktion er &41,&42 eller &44. Det vil sige, at bit 6 står for funktion. Lo-byte (1, 2 eller 4) angiver på sædvanlig vis, hvilken slags type funktionen resulterer i.

Alle funktionsnavne er knyttet til hinanden via indeksering. I adresse &AE04/5 (664/6128: &ADEB/C) står adressen på første element. Et funktionsnavn's værdi består af 2 bytes, der peger på det sted i BASIC-programmet, hvor funktionsdefinitionen er anført.

RAM-adresserne for de første elementer i hvert startbogstav står i adresserne &ADD0 til &AE03 (for 664/6128 : &ADB7 til &ADEA), hvor der optages 2 bytes for hvert bogstav. Når man benytter denne pointer, skal man huske, at det ikke er selve adressen, men derimod forskellen mellem adressen og startadressen i tabellen, der skal angives &AE85/6 (664/6128: &AE68/9).

## 9.3. XREF (Cross REFerence)

XREF kommandoens syntaks svarer til DUMP's. De to kommandoer ligger i det samme program, fordi store dele af programmet, er ens for begge kommandoer. Som ved DUMP findes de ønskede variabler i tabellen via løkkerne NEXBUC og GLEBUC.

I begyndelsen af programmet benytter XREF en særlig interessant systemrutine, for at sande alle variablerne i tabellen. På adresse &E8FF (6128: &E9B9 / 664: &E9BE) står rutinen BADOBC, der trinvis gennemløber et BASIC-program. Som noget helt specielt kalder den en anden rutine, når starten på en linie er fundet, så linien f.eks. kan undersøges. Adressen på den rutine, der skal foretage undersøgelsen, flyttes ved kald af BADOBC til BC-registeret. I XREF-programmet er den først adresse for INITVA-rutinen og senere for SUCHVA-rutinen.

For at forstå SUCHVA-rutinen, må vi beskæftige os lidt med opbygningen af et BASIC-program, og specielt med de i programmet benyttede variabler.

Foran hver variabel er der anført et kendetal. Dette tal angiver variabeltypen og om typekendetegnet er taget med (%, $ eller !). Her gælder at:

02 integer med %
03 streng med $
04 real med !
0B integer
0C streng
0D real

Efter kendetallet følger forskellen mellem tabellens startadresse og variablernes adresser. Efter denne adressepointer følger selve navnet, hvor bit 7 som sædvanligt er sat i sidste bogstav.

SUCHVA tester først, om det drejer sig om et kendetal. Hvis ja, så sammenlignes navnet, og kendetallet konverteres for sammenligning. Ved overensstemmelse udskrives BASIC-linienummeret for den aktuelle linie.

Til de programmører, der ikke er i besiddelse af Assembleren fra bogen "MASCHINENSPRACHE MIT DEN CPCs" eller en anden, følger der her en BASIC-loader til programmet. Implementeringen af de nye kommandoer med RSX sker via CALL &A000. Derefter står DUMP og XREF til disposition, på lige fod med de sædvanlige kommandoer.

```
 10 ' XREF OG DUMP FOR 464
 20 ' RSX IMPLEMENTERING MED CALL &A000
 30 FOR I=&A000 TO &A173
 40 READ A$:W=VAL("&H"+A$)
 50 S=S+W:POKE I,W:NEXT
 60 IF S <> 49982 THEN PRINT "FEJL I DATALINIER":END
 70 PRINT "OK!":END
 80 DATA 01,0F,A0,21,20,A0,CD,D1
 90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
120 DATA FC,A6,0F,A0,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,01,D0,5F,03
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,FF,E8,2A,75,AE
170 DATA 06,41,0E,5A,CD,4A,DD,B7
180 DATA 28,31,23,CD,37,DD,C0,7E
190 DATA CD,71,FF,38,03,C3,7B,D0
200 DATA 47,4F,CD,3F,DD,FE,2D,20
210 DATA 0C,CD,3F,DD,CD,71,FF,30
220 DATA EC,4F,CD,3F,DD,E5,CD,83
230 DATA A0,E1,CD,55,DD,38,D8,22
240 DATA 75,AE,C9,79,90,38,D6,78
250 DATA 04,C5,CD,DB,D5,7E,23,66
260 DATA 6F,B4,20,06,C1,79,B8,30
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,56,C3,F1,17,30
300 DATA EE,3E,20,CD,56,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,C1,B7
330 DATA ED,42,22,36,A0,C5,10,FD
340 DATA A0,CD,FF,E8,CD,3C,C4,CD
350 DATA 4E,C3,C1,E1,18,AF,E5,CD
```

```
360 DATA 43,E9,D1,FE,02,D8,FE,0E
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,3F,DD,CD
390 DATA B3,D6,E1,18,E1,E5,CD,43
400 DATA E9,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,3F,DD,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,36,AE,7E
450 DATA 23,66,6F,CD,79,EE,3E,20
460 DATA CD,56,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,56,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,4B,FF,E1,CD
510 DATA 36,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,56,C3,3E,20,CD,56
540 DATA C3,E1,F1,C9
```

***

```
 10 ' XREF OG DUMP FOR 664
 20 ' RSX IMPLEMENTERING MED CALL &A000
 30 FOR I=&A000 TO &A173
 40 READ A$:W=VAL("&H"+A$)
 50 S=S+W:POKE I,W:NEXT
 60 IF S <> 50380 THEN PRINT "FEJL I DATALINIER":END
 70 PRINT "OK!":END
 80 DATA 01,0F,A0,21,20,A0,CD,D1
 90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
120 DATA 38,A0,3E,0D,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,E1,D0,18,D8
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,BE,E9,2A,58,AE
170 DATA 06,41,0E,5A,CD,3C,DE,B7
180 DATA 28,31,23,CD,2A,DE,C0,7E
190 DATA CD,92,FF,38,03,C3,DA,D0
200 DATA 47,4F,CD,31,DE,FE,2D,20
210 DATA 0C,CD,31,DE,CD,92,FF,30
220 DATA EC,4F,CD,31,DE,E5,CD,83
230 DATA A0,E1,CD,46,DE,38,D8,22
```

```
240 DATA 58,A1,C9,79,90,38,D6,78
250 DATA 04,C5,CD,1C,D6,7E,23,66
260 DATA 6F,B4,20,06,C1,79,B8,30
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,A3,C3,FE,17,30
300 DATA EE,3E,20,CD,A3,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,C1,B7
330 DATA ED,42,22,36,A0,C5,01,FD
340 DATA A0,CD,BE,E9,CD,75,C4,CD
350 DATA 9B,C3,C1,E1,18,AF,E5,CD
360 DATA 02,EA,D1,FE,02,D8,FE,01
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,31,DE,CD
390 DATA EF,D6,E1,18,E1,E5,CD,02
400 DATA EA,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,31,DE,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,1D,AE,7E
450 DATA 23,66,6F,CD,49,EF,3E,20
460 DATA CD,A3,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,A3,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,6C,FF,E1,CD
510 DATA DA,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,A3,C3,3E,20,CD,A3
540 DATA C3,E1,FE,C9
```

***

```
 10 ' XREF OG DUMP FOR 6128
 20 ' RSX IMPLEMENTERING MED CALL &A000
 30 FOR I=&A000 TO &A173
 40 READ A$:W=VAL("&H"+A$)
 50 S=S+W:POKE I,W:NEXT
 60 IF S <> 50608 THEN PRINT "FEJL I DATALINIER":END
 70 PRINT "OK!":END
 80 DATA 01,0F,A0,21,20,A0,CD,D1
 90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
```

```
120 DATA 20,A0,0F,A0,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,01,D0,D1,00
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,B9,E9,2A,58,AE
170 DATA 06,41,0E,5A,CD,37,DE,B7
180 DATA 28,31,23,CD,25,DE,C0,7E
190 DATA CD,92,FF,38,03,C3,D7,D0
200 DATA 47,4F,CD,2C,DE,FE,2D,20
210 DATA 0C,CD,2C,DE,CD,92,FF,30
220 DATA EC,4F,CD,2C,DE,E5,CD,83
230 DATA A0,E1,CD,41,DE,38,D8,22
240 DATA 58,AE,C9,79,90,38,D6,78
250 DATA 04,C5,CD,19,D6,7E,23,66
260 DATA 6F,B4,20,06,C1,79,B8,30
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,A0,C3,F1,17,30
300 DATA EE,3E,20,CD,A0,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,CE,B7
330 DATA ED,42,22,36,A0,C5,01,FD
340 DATA A0,CD,B9,E9,CD,72,C4,CD
350 DATA 98,C3,C1,E1,18,AF,E5,CD
360 DATA FD,E9,D1,FE,02,D8,FE,0E
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,2C,DE,CD
390 DATA EC,D6,E1,18,E1,E5,CD,FD
400 DATA E9,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,2C,DE,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,1D,AE,7E
450 DATA 23,66,6F,CD,44,EF,3E,20
460 DATA CD,A0,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,A0,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,6C,FF,E1,CD
510 DATA D5,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,A0,C3,3E,20,CD,A0
540 DATA C3,E1,F1,C9
```

# 10. GENERERING AF PROGRAMLINIER I BASIC

En kommando, der indtil nu har manglet i alle BASIC-dialekter, er en som kan skabe nye linier programmæssigt. Det er en sådan kommando, der kan give helt nye muligheder for programmering i BASIC.

Findes der en kommando, hvormed man i et program kan skabe nye linier?

Hvis det er muligt, er det nærliggende at forestille sig, at et program selv kan fremstille et program, der selv kan fremstille ..... o.s.v.

Hvorfor overhovedet begynde på at lære at programmere selv, når computerne kan selv?

Så langt er udviklingen heldigvis ikke nået. Alligevel er der nogle interessante muligheder med "programmeringskommandoen". Men lad os nu kigge lidt på selve kommandoen.

Ved indtastning af BASIC-linier direkte i skærmeditoren, gemmes disse som strenge i en buffer, der tømmes når der tastes RETURN. Fortolkeren går i gang med at oversætte linien. Herved konverteres den til intern behandling. D.v.s. at f.eks. kommandoer ændres til de såkaldte TOKENs, og lagres i den korte form. Til slut indføjes linien i hovedprogrammet på en position, der er afhængig af linienummeret. Alle linier med et større nummer forskydes tilsvarende opefter.

For at virkeliggøre ideen, vil vi benytte de samme rutiner, som blev benyttet af fortolkeren i ovenstående gennemgang. Adresserne på de strenge, der skal "skabes", skal af brugeren løbende overgives programmet. Ved overflytningen af kun een værdi, står denne efter kald med CALL, eller også med RSX-udvidelser i DE-registeret. DE indeholder således STRING-descriptor-adressen for strengen, der skal konverteres. Herefter læses streng-adressen og flyttes til HL-registeret. Med CHRSKP-rutinen ignoreres evt. mellemrum og cursorbevægelser (HT og LF). Hvis der ikke er fundet nogen NULL-byte, testes der med TESTER for et linienummer i begyndelsen af linien. Hvis svaret er bekræftende, oversættes linien og indføjes med ASSEMB. Liniens slutning registreres via NULL-bytes. I det følgende findes assemblerlistning og en BASIC-loader til det netop beskrevne program.

```
A000    10      ; LINER
A000    20      ; FREMSTILNING AF basic-
                  LINIER
A000    30      ; I DIREKTE MODE
A000    40      ; I ET PROGRAM, HVIS LINIE-
                  NUMMER
A000    50      ; > AKTUELT
```

```
A000             60                          ; A$=LINIE I ASCII-FORMAT,
                                               ENDE=
A000             70                          ; NULL-BYTE
A000             80                          ; FORMAT : CALL &A000,@A$
A000             90
A000             100             ORG   &A000
A000             110                   ; CPC 6128 ; 464 , 664
A000             120     CHKSKP  EQU   &DE4D ; &DD61 , &DE52
A000             130     TESTER  EQU   &EECF ; &EE04 , &EED4
A000             140     ASSEMB  EQU   &E7A5 ; &E6C6 , &E7AA
A000 DF          150             RST   &18 ; FAR CALL
A001 0000        160             DW    VEKTOR ; DA BASIC ROM SKAL
A003 C9          170             RET   ; VÆRE AKTIV
**** LINIE 160 :         VEKTOR=&A004
A004 0000        180     VEKTOR  DW    START
A006 FD          190             DB    253 ; LO ROM OFF, UP ROM ON
**** LINIE 180 :         START=&A007
A007 EB          200     START   EX    DE,HL ; ADRESSE TIL HL
A008 23          210             INC   HL ; STRENGLÆNGDE
A009 5E          220             LD    E,(HL) ; LO-BYTE STRENG-
                                       ADRESSE
A00A 23          230             INC   HL
A00B 56          240             LD    D,(HL) ; HI-BYTE STRENG-
                                       ADRESSE
A00C EB          250             EX    DE,HL ; STRENGADRESSE TIL HL
A00D CD4DDE      260             CALL  CHRSKP
A010 B7          270             OR    A
A011 C8          280             RET   Z
A012 CDCFEE      290             CALL  TESTER
A015 D0          300             RET   NC
A016 CDA5E7      310             CALL  ASSEMB
; OVERSÆT LINIE OG INDFØJ
; HL SKAL PEGE PÅ FØRSTE BYTE I ASCII-LINIE
A019 C9          320             RET   ; FÆRDIG !!!
```

```
PROGRAM: LINIER
START:   &A000
SLUT:    &A019
LÆNGDE:  001A
FEJL:    0
```

---

VARIABELTABEL:

---

CHRSKP DE4D  TESTER EECF  ASSEMB E7A5  VEKTOR A004
START  A007

---

```
 10 REM BASIC-LOADER FOR LINIER
 20 FOR I=&A000 TO &A019
 30 READ A$:W=VAL("&H"+A$)
 40 S=S+W:POKE I,W:NEXT
 50 IF S >< 4275 THEN PRINT "FEJL I DATALINIER":END
 60 ' 464 : IF S <> 4123 THEN .....
 70 ' 664 : IF S <> 4290 THEN .....
 80 PRINT "OK!!":END
 90 DATA DF,04,A0,C9,07,A0,FD,EB
100 DATA 23,5E,23,56,EB,CD,4D,DE
110 ' 664 : .., .., .., .., .., 52,DE
120 DATA B7,C8,CD,CF,EE,D0,CD,A5
130 ' 464 : .., .., .., 04,EE, .., ..,C6
140 ' 664 : .., .., .., D4,EE, .., ..,AA
150 DATA E7,C9
160 '464:  E6,C9
170 '664:  E7,C9
```

Til demonstration af kommandoens funktion følger et lille program:

```
10 MEMORY &9FFF : REM ER LINIER LOADED ?
20 ZEINU=100
30 Z$=STR$(ZEINU)+"REM DETTE ER LINIEN"+STR$(ZEINU)
40 Z$=Z$+CHR$(0):REM SLUTMARKERING
50 CALL &A000,@Z$
60 ZEINU=ZEINU+10
70 IF ZEINU<210 THEN 30
80 Z$=STR$(ZEINU)+"LIST"+CHR$(0)
90 CALL &A000,@Z$
```

Der er følgende indskrænkninger i brugen af programmet:

— Linienummeret på den linie, der skal skabes, SKAL være højere end det linienummer, hvori CALL-kommandoen ligger.

— Kommandoen må IKKE stå inde i en FOR-NEXT-løkke eller en WHILE-WEND-løkke.

Hvis ikke ovenstående overholdes, kan BASIC-programmet ikke finde "hjem" igen efter udført CALL og variablernes indhold tabes ligeledes. Årsagen er, at BASIC-programmet jo forskydes ved indføjning af den nye linie.

Den nemmeste måde at bruge kommandoen på, er sikkert til at lave DATA-linier i f.eks. en database. Ved hjælp af PROGRAM-GENERATOREN, kan man gemme DATA direkte i et program.

I bogen "Maschinensprache fur 464, 664, 6128" fra DATA BECKER er der et program, der skaber en kommando til generering af programmer. En professionel udnyttelse af kommandoen kunne lyde således:

Et softwarefirma sælger økonomisystemer. For at kunne tilfredsstille den enkelte kundes særlige behov og ønsker, laves der et softwareudviklingsprogram, der tager højde for alle tilpasningsmuligheder. Efter at de særlige ønsker og behov er tastet ind i programmet, kan udviklingsystemet generere et individuelt program.

# 11. GRAFIK HARD-COPY

For at kunne få de med 3-D-funktionsplotteren fremstillede tegninger ud på papir, har vi lavet et hard-copy-program. Programmet kan køre uden modifikationer på en EPSON FX-80 printer (og kompatible). For at kunne køre på andre printere, skal den styresekvens, der sætter printeren i grafik-mode ændres, alt forudsat at der er tale om en 8-punkts matrixprinter.

Til kopieringen udlæses Amstrad's Video-RAM i adresserne &C000 til &FFFF i mode 2. Skærmhukommelsen er på CPC opbygget på den gammelkendte mærkværdige måde. Som hos de fleste andre computere, gælder det grundlæggende, at en byte i hukommelsen svarer til et bestemt antal punkter på skærmen (8 i MODE 2). Desværre opstår der et problem, fordi printerens 8 punkter ligger lodret placeret og skærmhukommelsens ligger ordnet ved siden af hinanden (vandret).

Lad os som et eksempel betragte det følgende specialtegn. Dette tegn skal udskrives på skærmen.

| | | | | | | | | KODE: |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = 00 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | = 40 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | = 68 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | = 186 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | = 68 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | = 40 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | = 16 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | = 254 |

PRINTERKODE: 17 41 85 19 85 41 17 0

Tegnet udskrives på skærmen med følgende sekvens:

```
MODE 2
FOR I=&C000 TO &F800 STEP &800
POKE I,A:NEXT
DATA 16,40,68,16,68,40,16,254
```

For udskrivning på printer, skal værdierne sendes kolonnevis :

```
PRINT#8,CHR$(27);"*"CHR$(1);CHR$(8);CHR$(0);
FOR I=0 TO 7: READ A
PRINT#8,CHR$(A);:NEXT
DATA 17,41,85,19,85,41,17,0
```

Den første PRINT#8-kommando er styresekvensen for EPSON FX-80, der sørger for udskrivning af 8 grafikkoder i bitmønster-mode. I DATA BECKER bogen "Grosse

Epson Druckerbuch" kan man finde alle styrekoderne til Epson-printerne og en del avanceret brug af printeren.

Hard-copy programmets opgave er, at konvertere hele skærmbilledet trinvis for at kunne viderebringe "kolonne-værdierne" til printerens skrivehovede. I BASIC er det faktisk en sag, der kan vare flere timer! Derfor kan der kun blive tale om at lave et sådant program i maskinkode.

Programmet udnytter nogle særdeles nyttige systemrutiner. De vil blive beskrevet i det følgende.

Med Amstrad-computere er det sværere end normalt at lave hard-copy programmet uden hardwaremæssige ændringer. Dette skyldes, at selvom konstruktørerne af maskinen har forsynet den med et Centronics interface, så har man af lidt uforståelige grunde kun belagt den med en 7-bit-port. Normalt vil den være på 8, idet en byte består af 8 bits og en printer styres byte-vis.

Så længe man kun printer tekster og listninger ud, mærker man ikke noget til mangelen. I en hard-copy vil man kunne "se den manglende bit", da den resulterer i en hvid streg for hver 8. punktlinie!

Programmet må altså tage højde for dette. Der skal udskrives 7 punktlinier, og en line-feed må kun være af 7 punktbredder. Men da Amstrad's skærmhukommelse er organiseret i vertikale enheder a 8 bytes og hvert tegn er 8 bits højt, er det nogle omfattende beregninger, der skal udføres, hvis 7 bits skal printes med en nogenlunde tilfredsstillende hastighed.

Skærmbilledet består af 25 linier a 8 punktlinier, d.v.s. 200 rækker af punkter. Skal disse bearbejdes 7 af gangen, er det 28 * 7 linier, der skal sendes. Problemet opstår til sidst, hvor der bliver 4 (200-28*7) rækker tilovers. De skal behandles for sig. Yderligere, består skærmen horisontalt af 640=&280 punkter.

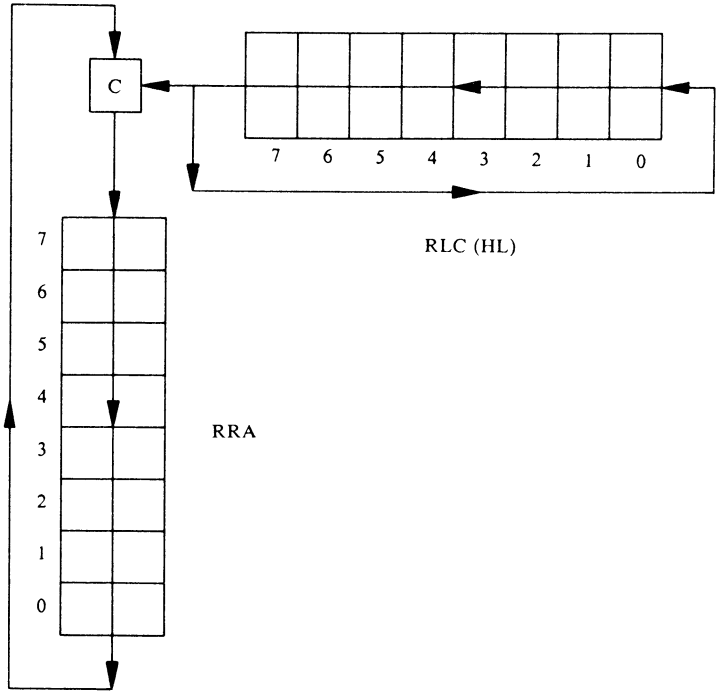
FX-80 (og kompatible) bliver ved opstart af skærmmode meddelt, hvor mange grafikbytes, der skal modtages. Lo-byte for &280 er &80. Ved denne byte er den 8. bit sat, d.v.s. at denne værdi ikke kan overføres til printeren. For at løse problemet uden større ændringer, bliver grafik-printeren kun tilkendt &27F grafik-bytes. Det betyder, at den sidste række aldrig vil fremkomme på en hard-copy. Men det er vel også til at leve med.

Men nu til selve programmet.

For udlæsning af de, til enhver tid, 7 under hinanden beliggende bytes, opbygges en tabel, hvori de syv aktuelle byte's adresser er lagret. Den 8. adresse i tabellen vil altid indeholde den første af de 7 næste bytes, der skal behandles. Tabellen bliver initialiseret via LABEL NEL1 for hver syvende linie. Her skal systemrutinen SCR NEXT LINE benyttes. Den forhøjer den til HL flyttede skærmadresse, således at den peger på den næste punktlinie på skærmen. I løkken LABEL NEBIT konverteres de 8 vandrette bits til det lodrette 7-bits format.

Begyndende med det sidste tabel-element (TABLET), loades byteadressen i HL-registeret. Via RLC (HL) roteres de enkelte bytes mod venstre, hvorved MSB (Most Significant Bit) også flyttes til CARRY.

Billedpunktet befinder sig således nu i CARRY og kan roteres til AKK med RRA. Så starter løkken forfra. Efter syv gennemløb indeholder AKK de syv højeste bits (punkter) i linien. Denne løkke er et fremragende eksempel på maskinkodekommandoers store styrke og hastigheder. Løkkens funktion kan studeres endnu en gang via følgende skitse:




Efter NEBIT-løkkens afslutning, roteres AKK-indholdet endnu en gang, således at vores fundne bitmatrix ikke gør brug af bit nr. 7, der jo ikke kan sendes. Til slut sendes den opbyggede byte til printeren.

Denne procedure gennemløbes ialt 8 gange for hver byte. Herved er hver enkelt bit trinvis blevet roteret i AKK. En sjov "bivirkning" af denne effekt lader sig afsløre ved et nærmere blik på skærmen. Det ser virkeligt ud som om bytene roterer på skærmen. For det meste, ses det kun i starten af en ny linie, idet programmet må vente på printeren.

Er alle 8 bytes blevet sendt, sættes hvert tabel-element på den næste byte, via løkken fra NEBY1, med hjælp fra systemrutinen SCR NEXT BYTE. Ved overflytningen af den aktuelle skærmadresse i HL-registeret, forhøjer SCR NEXT LINE værdien i HL, således at den næste byte adresseres i samme linie. Via byte-tælleren C, hoppes der igen til konvertering og udskrivning, så længe slut ikke er nået.

Er slutningen på en linie nået, så meddeles det via tælleren E, om der skal trykkes flere linier. Hvis det er således, hoppes der til LABEL NELINE. Her starter hele møllen forfra. Er det den sidste linie, der skal printes (LD A,E; CP 1), så sættes antallet af bits, der ligger under hinanden, til 4, og tabellen udfyldes kun fra midten med 4 startadresser.

Her har vi yderligere gjort brug af et lille programmeringstrick. Efter at tabellens definitionsværdier for den sidste linie er blevet sat med LD DE,TABMIT og LD B,4, så skal kommandoen LD DE,TANANF naturligvis springes over. Det sker normalt med JR-kommando. Det er dog at bruge en byte med værdien &21 i stedet for JR-kommandoen. Computeren fortolker &21 som Opcode for kommandoen LD HL,nn. Dermed er de følgende 2 bytes knyttet til denne kommando. Den 3. byte er Hi-byte for TABANF, d.v.s. &A0. &A0 står for kommandoen AND B. De to kommandoer er ikke til gene for programudførelsen, og ændrer heller ikke på noget vigtigt register-indhold. Efter de to kommandoers udførelse, fortsættes der med den regulære udførelse via kommandoen LD HL,(TABEND). Den ønskede effekt, nemlig at der springes over kommandoen LD DE,TABEND, er hermed opnået.

Der er endnu to underprogrammer, der skal nævnes: TABOUT og PRINT.

TABOUT står for tabeludlæsning. Dermed menes tabeller med styrekommandoer for printer. Den første styresekvens er PRINT. Den sætter Line-feed til 7/72 tomme og udfører Line-feed. For at kunne sende denne styresekvens til printeren, bliver adressen for den første byte i rækken loaded i HL og TABOUT kaldes. TABOUT kan detektere slutningen på en sekvens, via en række null-bytes.

Yderligere findes der sekvenserne PRLIIN og PRREIN. PRLIIN (Printer Line Init) sender en styresekvens, der sætter printeren til 639 bytes (een linie) i 8-bits mønstermode. PRREIN (Printer Re-init) reset'er til standard-mode.

For at hard-copy programmet også kan tilpasses andre printere, er der reserveret 4 bytes i hver sekvens. Er man i besiddelse af en printer, som ikke udfører line-feed, kan man erstatte kode 24 i PRLIIN-sekvensen med kode 10 (LF).

Print-rutinen står for udskrivningen på printer. Først formindskes tælleren PRTCOU og der returneres ved værdien 0. Herved sendes der kun 639 af de 640 bytes. Med MCPRBU testes der om printeren er "BUSY" (i færd med at udføre en sekvens). Hvis bekræftende, testes der videre. Ellers sendes byten indeholdt i AKK til printeren med MCPRCH.

```
A000             10                      ; HARDCOPY AF H.D. 18/10/85
A000             20                      ; FOR EPSON FX80 + KOMPATIBLE
A000             30                      ; KØRER PÅ CPC 464, 664 OG 6128
A000             40                      ; UDEN MODIFIKATIONER
A000             50              ORG     &A000
A000             60      SCGELO  EQU     &BC0B ; SCREEN GET LOCATION
A000             70      SCNELI  EQU     &BC26 ; SCREEN NEXT LINE
A000             80      SCNEBY  EQU     &BC20 ; SCREEN NEXT BYTE
A000             90      MCPRCH  EQU     &BD2B ; MC PRINT CHARACTER
A000             100     MCPRBU  EQU     &BD2E ; MC PRINTER BUSY ?
A000             110
A000  210000     120             LD      HL,PRINIT ; PRINTER INIT
A003  CD0000     130             CALL    TABOUT
A006  CD0BBC     140             CALL    SCGELO
A009  57         150             LD      D,A
A00A  1E00       160             LD      E,0
A00C  19         170             ADD     HL,DE ; PUNKTS ADR. ØVERST
                                         VENSTRE
A00D  220000     180             LD      (TABEND),HL
A010  1E1D       190             LD      E,29 ; LINIE-TÆLLER
A012  3E07       200             LD      A,7
A014  320000     210             LD      (BITANZ),A ; BITS PR. PRINT-
                                         LINIE
A017  D5         220     NELINE  PUSH    DE ; GEM LINIE-TÆLLER
A018  0608       230             LD      B,8 ; ÆNDRING AF LINIE START-
                                         ADRESSE
A01A  7B         240             LD      A,E ; HVIS IKKE SIDSTE LINIE
A01B  FE01       250             CP      1
A01D  20FE       260             JR      NZ,OK ; IKKE, ALT IORDEN
A01F  3E04       270             LD      A,4 ; ELLERS KUN 4 BITS PR.
                                         LINIE
A021  320000     280             LD      (BITANZ),A ; BITS PR. LINIE
A024  110000     290             LD      DE,TABMIT ; FYLD KUN 4
                                         ELEMENTER
A027  0604       300             LD      B,4 ; I TABELLEN
A029  21         310             DB      &21 ; "ØDELÆG" NÆSTE
                                         KOMMANDO.
****  LINIE 260 :     OK=&A02A
A02A  110000     320     OK      LD      DE,TABANF
A02D  2A0000     330             LD      HL,(TABEND) ; HL ER DET NYE
                                         FØRSTE
A030  EB         340     NELI1   EX      DE,HL ; ELEMENT I TABELLEN
A031  73         350             LD      (HL),E
A032  23         360             INC     HL
A033  72         370             LD      (HL),D
A034  23         380             INC     HL
A035  EB         390             EX      DE,HL
```

```
A036 CD26BC 400          CALL SCNELI ; NÆSTE LINIE-ADRESSE
A039 10F5   410          DJNZ NELI1 ; HENTES OG LAGRES
A03B 210000 420          LD   HL,PRLIIN ; BIT-MØNSTER-
                              MODE
A03E CD0000 430          CALL TABOUT ; AKTIVERES
A041 217F02 440          LD   HL,&27F ; MEN SEND KUN 639
                              BYTES
A044 220000 450          LD   (PRTCOU),HL
A047 015008 460          LD   BC,&850 ; C=BYTETÆLLER=80
A04A C5     470 NEBY     PUSH BC
A04B 3A0000 480          LD   A,(BITANZ) ; ANTAL BITS FOR
                              PRINT
A04E 47     490          LD   B,A
A04F 97     500          SUB  A ; SLET AKK
A050 210000 510          LD   HL,TABLET ; BEGYND MED
                              SIDSTE
A053 56     520 NEBIT    LD   D,(HL) ; ELEMENT
A054 2B     530          DEC  HL
A055 5E     540          LD   E,(HL) ; LÆS BYTEADRESSE
A056 2B     550          DEC  HL
A057 EB     560          EX   DE,HL
A058 CB06   570          RLC  (HL) ; ROTER BYTE
A05A 1F     580          RRA  ; ROTER CARRY I AKK
A05B EB     590          EX   DE,HL
A05C 10F5   600          DJNZ NEBIT ; NÆSTE BIT
A05E 1F     610          RRA  ; BENYT IKKE BIT 7
A05F CD0000 620          CALL PRINT ; UDSKRIV
A062 C1     630          POP  BC
A063 10E5   640          DJNZ NEBY ; 8 GANGE PR. BYTE
A065 210000 650          LD   HL,TABANF ; 7 TABEL-
                              ELEMENTER,
A068 0607   660          LD   B,7 ; HVOR HVER FORHØJES MED
                              EN
A06A 5E     670 NEBY1    LD   E,(HL); BYTE MOD HØJRE
A06B 23     680          INC  HL
A06C 56     690          LD   D,(HL)
A06D 2B     700          DEC  HL
A06E EB     710          EX   DE,HL
A06F CD20BC 720          CALL SCNEBY
A072 EB     730          EX   DE,HL
A073 73     740          LD   (HL),E
A074 23     750          INC  HL
A075 72     760          LD   (HL),D
A076 23     770          INC  HL
A077 10F0   780          DJNZ NEBY1
A079 0608   790          LD   B,8
A07B 0D     800          DEC  C ; ENDNU IKKE SLUT PÅ LINIE?
A07C 20CC   810          JR   NZ,NEBY ; SÅ HENT NÆSTE
                              BYTE
```

```
A07E D1       820            POP  DE
A07F 1D       830            DEC  E ; SIDSTE LINIE ?
A080 2095     840            JR   NZ,NELINE ; NEJ, SÅ NÆSTE
A082 210000   850            LD   HL,PRREIN ; PRINTER RE-INIT
A085 18FE     860            JR   TABOUT ; SÅ SLUT
A087          870            ;
****  LINIE 130 :  TABOUT=&A087
****  LINIE 430 :  TABOUT=&A087
****  LINIE 860 :  TABOUT=&A087
A087 7E       880  TABOUT    LD   A,(HL) ; UDSKRIV TABEL PÅ
                                  ADRESSE
A088 FE00     890            CP   0 ; HL TIL NULL-BYTE
A08A C8       900            RET  Z
A08B 23       910            INC  HL
A08C E5       920            PUSH HL
A08D CD0000   930            CALL PRINT
A090 E1       940            POP  HL
A091 18F4     950            JR   TABOUT ; NÆSTE BYTE
A093          960            ;
****  LINIE 620 :  PRINT=&A093
****  LINIE 930 :  PRINT=&A093
A093 2A0000   970  PRINT      LD   HL,(PRTCOU) ; FORMINDSK
                                  BYTE-TÆLLER
A096 2B       980            DEC  HL
A097 220000   990            LD   (PRTCOU),HL ; OG GEM DEN IGEN
A09A 4F       1000           LD   C,A ; MELLEMLAGRING AF
                                  KARAKTER
A09B 7C       1010           LD   A,H ; TEST FOR BYTE-TÆLLER = 0
A09C B5       1020           OR   1
A09D C8       1030           RET  Z
A09E 79       1040           LD   A,C ; HVIS NEJ, UDLÆS AKK
A09F CD2EBD   1050 WAIT      CALL MCPRBU ; PRINTER BUSY ?
A0A2 38FB     1060           JR   C,WAIT ; JA, FORTSÆT VENTNING
A0A4 CD2BBD   1070           CALL MCPRCH ; UDLÆS TEGN
A0A7 C9       1080           RET
A0A8          1090
****  LINIE 120 :  PRINIT=&A0A8
A0A8 1B       1100 PRINIT    DB   27 ; ESC
A0A9 31       1110           DM   "1" ; 7/72 TOMME LF
A0AA 0D00     1120           DW   &000D ; CR OG 0-BYTE
A0AC 0000     1130           DW   0 ; RESERVERET PLADS FOR
A0AE 0000     1140           DW   0 ; ÆNDRINGER
A0B0          1150
****  LINIE 420 :  PRLIIN=&A0B0
A0B0 0D       1160 PRLIIN    DB   13
A0B1 18       1170           DB   24 ; CANcel EVT. VIA 10=LF
A0B2 1B       1180           DB   27
A0B3 2A       1190           DM   "*" ; SÆT BITMØNSTER-MODE
A0B4 01       1200           DB   1 ; MODE 1
```

```
A0B5 7F02      1210            DW   &027F
A0B7 00        1220            DB   0 ; 0-BYTE
A0B8 0000      1230            DW   0 ; RESERVERET PLADS FOR
A0BA 0000      1240            DW   0 ; ÆNDRINGER
A0BC           1250
**** LINIE 850 :    PRREIN=&A0BC
A0BC 1B        1260 PRREIN     DB   27 ; ESC
A0BD 40        1270            DM   "@" ; RESET PRINTER
A0BE 0D00      1280            DW   &000D ; CR OG 0-BYTE
A0C0 0000      1290            DW   0 ; RESERVERET PLADS FOR
A0C2 0000      1300            DW   0 ; ÆNDRINGER
A0C4           1310
**** LINIE 210 :    BITANZ=&A0C4
**** LINIE 280 :    BITANZ=&A0C4
**** LINIE 480 :    BITANZ=&A0C4
A0C4 07        1320 BITANZ     DB   7
**** LINIE 450 :    PRTCOU=&A0C5
**** LINIE 970 :    PRTCOU=&A0C5
**** LINIE 990 :    PRTCOU=&A0C5
A0C5 7F02      1330 PRTCOU     DW   &27F
**** LINIE 320 :    TABANF=&A0C7
**** LINIE 650 :    TABANF=&A0C7
A0C7           1340 TABANF     DS   6
**** LINIE 290 :    TABMIT=&A0CD
A0CD           1350 TABMIT     DS   7
**** LINIE 510 :    TABLET=&A0D4
A0D4           1360 TABLET     DS   1
**** LINIE 180 :    TABEND=&A0D5
**** LINIE 330 :    TABEND=&A0D5
A0D5           1370 TABEND     DS   2
```

PROGRAM: HARD-COPY
START: &A000
SLUT: &A0D6
LÆNGDE: 00D7
FEJL: 0

---

VARIABELTABEL:

---

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| SCGELO | BC0B | SCNELI | BC26 | SCNEBY | BC20 | MCPRCH | BD2B |
| MCPRBU | BD2E | NELINE | A017 | OK | A02A | NELI1 | A030 |
| NEBY | A04A | NEBIT | A053 | NEBY1 | A06A | TABOUT | A087 |
| PRINT | A093 | WAIT | A09F | PRINIT | A0A8 | PRLIIN | A0B0 |
| PRREIN | A0BC | BITANZ | A0C4 | PRTCOU | A0C5 | TABANF | A0C7 |
| TABMIT | A0CD | TABLET | A0D4 | TABEND | A0D5 | | |

---

```
 10 REM HARD-COPY FOR ALLE AMSTRAD-COMPUTERE
 20 REM KALDES MED CALL &A000 I MODE 2
 30 FOR I=&A000 TO &A0C6
 40 READ A$:W=VAL("&H"+A$)
 50 S=S+W:POKE I,W:NEXT
 60 IF S <> 19650 THEN PRINT "FEJL I DATALINIER!":END
 70 PRINT"OK!":END
 80 DATA 21,A8,A0,CD,87,A0,CD,0B
 90 DATA BC,57,1E,00,19,22,D5,A0
100 DATA 1E,1D,3E,07,32,C4,A0,D5
110 DATA 06,08,7B,FE,01,20,0B,3E
120 DATA 04,32,C4,A0,11,CD,A0,06
130 DATA 04,21,11,C7,A0,2A,D5,A0
140 DATA EB,73,23,72,23,EB,CD,26
150 DATA BC,10,5F,21,B0,A0,CD,87
160 DATA A0,21,7F,02,22,C5,A0,01
170 DATA 50,08,C5,3A,C4,A0,47,97
180 DATA 21,D4,A0,56,2B,5E,2B,EB
190 DATA CB,06,1F,EB,10,F5,1F,CD
200 DATA 93,A0,C1,10,E5,21,C7,A0
210 DATA 06,07,5E,23,56,2B,EB,CD
220 DATA 20,BC,EB,73,23,72,23,10
230 DATA F1,06,08,0D,20,CC,D1,1D
240 DATA 20,95,21,BC,A0,18,00,7E
250 DATA FE,00,C8,23,E5,CD,93,A0
260 DATA E1,18,F4,2A,C5,A0,2B,22
270 DATA C5,A0,4F,7C,B5,C8,79,CD
280 DATA 2E,BD,38,FB,CD,2B,BD,C9
290 DATA 1B,31,0D,00,00,00,00,00
300 DATA 0D,18,1B,2A,01,7F,02,00
310 DATA 00,00,00,00,1B,40,0D,00
320 DATA 00,00,00,00,07,7F,02
```

# 12. TIDEN ER INDE .... I CPC

For at læseren ikke skal glemme tiden når der programmeres, har vi lavet et software-ur. Når det bruges, bliver man sparet for megen ærgelse med familien og vennerne. Det er tillige et udmærket plus i mange "rigtige" programmer.

Et sådant program kan kun skrives med hjælp fra interruptstyringen. Programmering med interrupt er vanskeligere end "normal" programmering i maskinsprog. Det er i forvejen vanskeligt at teste programmer skrevet i maskinkode, men når det drejer sig om interruptprogrammering, er det næsten en umulighed, da et sådant program faktisk udføres via "tiden".

Interruptprogrammering åbner uanede muligheder. Faktisk, kan computeren slet ikke fungere uden interrupt. Hovedformålet med interrupt er aflæsningen (scanning) af tastaturet. Interrupts udført fra BASIC afvikles naturligvis også via den interne interrupt.

Lad os starte med begyndelsen. I enhver computer er der et stykke quartz, der via en bestemt frekvens styrer og synkroniserer hele forløbet. Dette stykke quartz er dermed det grundlæggende interne ur i computeren. I mange computere er der flere sådanne taktgivere. Via en af disse, i Amstrad via Gate Array, sættes Z80 processorens IRQ (Interrupt Request) pin i regelmæssige intervaller til Lo (LOW). Herved vil et, til enhver tid, kørende maskinkodeprogram hoppe til en adresse, afhængig af interrupt-mode.

Z80-processoren bliver i Amstrad styret via interrupt-mode 1. En IRQ afstedkommer herved enten en RST &38 eller en CALL &0038. IRQ forekommer i Amstrad 300 gange i sekundet. Hvis ikke interrupt er slået fra med DI, bliver den aktuelle adresse i programudførelsen lagt i STACK, og der hoppes til adresse &38. Her sker der endnu et spring til adresse &B989 (6128: &B941/ 664:&B941) d.v.s. i RAM, hvor den egentlige interruptrutine kan begynde.

Herfra fortsættes i nederste ROM og der springes til Interrupt-Service-Rutinen i adresse &00B1. Her klares tastaturafsøgningen og forhøjelsen af BASIC-variablen TIME.

En anden ROM-rutine sørger for betjeningen af BASIC-interrupts. Det er her hele styringen af sound-chippen foregår via interruptrutiner.

For at kunne implementere en brugerdefineret interruptrutine, skal der lægges en patch over hoppet til service-rutinen, der skal kalde vores rutine. I slutningen af denne rutine, skal der naturligvis ske en forgrening til den egentlige interrupt-rutine; d.v.s. adresse &00B1.

Med disse forudsætninger, kan vi udvikle vores egen rutine for implementering. Fra det tidspunkt, hvor den aktiveres, vil den blive kaldt hver 1/300 sekund uden at øve nævneværdig indflydelse på de øvrige processer, som foregår i computeren.

Det betyder for vort softwareur, at det vil fortsætte med at gå, selv i direkte mode. Lige så længe, computeren er tændt, vil tiden blive vist på skærmen, på en ønsket position. Lad os beskæftige os lidt med selve rutinen.

I starten af programmet ligger den lille initialiseringsrutine, der får CALL-kommandoen for Interrupt-Service-Rutinen til at springe til vores egen rutine. De følgende kommandoer ændrer init-rutinen, så den ved fornyet kald kan forlade rutinen igen.

Rutinen begynder ved label START. Det første, der sker her, er at retur-adressen for Interrupt-Service-Rutinen lægges i STACK. Det betyder at rutinen derefter kan afsluttes med RET, og at der automatisk fortsættes ved adresse &00B1.

Den første tæller COUNT vil derefter, ved hvert kald, blive formindsket med 1. Herved opnåes, at den nye tid kun udskrives hver 300. gang. På den måde undgår man unødig tidsspilde og forsinkelse af andre processer. Hvis tælleren er lig med 0, vil den blive tilskrevet værdien 300 påny, for de næste gennemløb. Og nu begynder det egentlige ur-program.

Lagring af tiden kræver 3 bytes. 1 byte for time-angivelse, 1 for minutter og 1 byte for sekunder. Den aktuelle tid gemmes i time-byten. Hver byte-værdi lagres i BCD-format. Det betyder, at hvert ciffer (decimalværdi) lagres i hold af 4 bits. BCD betyder Binary Coded Decimal. P.g.a. de specielle egenskaber hos det hexadecimale talsystem, kan man også skrive følgende. Decimal 16 er i BCD-format &16. BCD-formen er i vort tilfælde hurtigere og mere effektiv end normal lagring af værdierne. Regning med 1-byte-BCD-tal er næppe langsommere end med normale tal, idet Z80 processoren har en speciel kommando til BCD-aritmetik, nemlig DAA. Udlæsning af et BCD-tal i forhold til et normalt tal, er meget nemmere, og derfor også hurtigere. Vi opnår at optage mindst muligt af den kostbare regnetid i computeren.

Især ved brug af interrupt-rutiner, er det væsentligt at medtage regnehastigheden som faktor.

Uret stilles et sekund frem ved at adressen for sekund-bytes i HL-registeret og det maksimale antal sekunder (60) lægges i hænderne på underprogrammet ZEIT. Underprogrammet ZEIT forhøjer, det på den angivne adresse beliggende BCD-tal, med 1. Derefter testes der for maksimalværdien (60). Er denne nået, slettes Carry-Flag, ellers sættes det. Er maksimalværdien ikke nået, hoppes der efter returnering til hovedrutinen, straks til udskrivning af den nye tid. Det er muligt fordi minut- og timeangivelsen kun skal ajourføres, når sekundværdien er nået op på 60 enheder.

Er maksimalværdien nået, vil ZEIT sætte byten til 0 igen, fordi der efter 59 sekunder, 59 minutter eller 24 timer skal startes forfra igen. Ved opnåelse af maksimalværdien i sekundbyten bliver HL loaded med adressen for minutter, og for at kunne forhøje den, bliver ZEIT kaldt igen. Er minutterne også nået op på den maksimale værdi, skal timerne også forhøjes. Senest på dette tidspunkt, er den aktuelle tid blevet sat og er klar til udlæsning.

Til dette job bliver POS loaded med skærmpositionen for udskrift af tiden. Via rutinen BCBYOU (BCd BYte OUt) udskrives timer, minutter og sekunder adskilt med koloner. Det er i forbindelse med BCBYOU, at fordelen med BCD-tal fremkommer. AKK loades med &30 (ASCII 0). Med RLD roteres et BCD-tal i de nederste 4 bits i AKK (HL peger hele tiden på den aktuelle byte). Herved befinder det aktuelle ASCII-ciffer sig hele tiden i AKK, hvorfra det kan udlæses. Udskrivningen sker med underprogrammet PRINT, der for hver udskrift beregner og gemmer positionen for den næste udskrivning. Her følger nu hele assembler-listningen:

```
A000            10                      ; SOFTWARE-UR FOR ALLE
                                          AMSTRAD'S
A000            20                      ; H.D. 22/9/85
A000            30
A000            40                      ; INITIALISERINGSRUTINE
A000  210000    50              LD      HL,START ; INTERRUPTRUTINE
A003  2251B9    60              LD      (&B951),HL ; PATCH
A006            70                      ; FOR 464 : LD (&B949),HL
A006  21B100    80              LD      HL,&B1
A009  2201A0    90              LD      (&A001),HL
A00C  C9        100             RET
A00D            110
A00D            120     WRITE   EQU     &BDD3 ; (464,664 OG 6128)
A00D            130     INTCOU  EQU     300
A00D            140     POSITI  EQU     &4700 ; MODE 2
A00D            150     DOPPUN  EQU     &3A
A00D            160
****  LINIE 50 :        START=&A00D
A00D  21B100    170     START   LD      HL,&B1 ; RETURADRESSE
A010  E5        180             PUSH    HL ; INTERRUPT-SERVICE-
                                          RUTINE
A011  2A0000    190             LD      HL,(COUNT) ; HENT TÆLLER
A014  2B        200             DEC     HL ; FORMINDSK SÅLEDES AT
                                          KUN
A015  220000    210             LD      (COUNT),HL
A018  7C        220             LD      A,H ; SEKUNDER (AF RESTEN)
                                          UDFØRES
A019  B5        230             OR      1 ; ALTSÅ: HVIS IKKE TÆLLER 0,
                                          SÅ
A01A  C0        240             RET     NZ ; FÆRDIG
A01B  212C01    250             LD      HL,INTCOU ; INITIALISER
                                          TÆLLER IGEN
A01E  220000    260             LD      (COUNT),HL
A021            270
A021  210000    280             LD      HL,SEC ; SEKUND-BYTEADRESSE
A024  0660      290             LD      B,&60 ; MAX 60 (BCD) SEKUNDER
A026  CD0000    300             CALL    ZEIT ; FORHØJ SEKUNDER
A029  38FE      310             JR      C,ANZEI ; ENDNU IKKE 60, SÅ VIS
A02B  CD0000    320             CALL    ZEIT ; FORHØJ MINUTTER
```

```
A02E 38FE    330            JR    C,ANZEI
A030 0624    340            LD    B,&24 ; MAKS. 24 TIMER
A032 CD0000  350            CALL  ZEIT ;
****  LINIE 310 :    ANZEI=&A035
****  LINIE 330 :    ANZEI=&A035
A035 210047  360  ANZEI     LD    HL,POSITI ; POSITION FOR
                                  UDSKRIFT
A038 220000  370            LD    (POS),HL
A03B 210000  380            LD    HL,STUND
A03E CD0000  390            CALL  BCBYOU ; UDLÆS BCD-FORMAT
                                  BYTE
A041 3E3A    400            LD    A,DOPPUN
A043 CD0000  410            CALL  PRINT
A046 CD0000  420            CALL  BCBYOU ; UDLÆS MINUTTER
A049 3E3A    430            LD    A,DOPPUN
A04B CD0000  440            CALL  PRINT
A04E CD0000  450            CALL  BCBYOU ; UDLÆS SEKUNDER
A051 C9      460            RET
A052         470
A052         480
; UNDERPROGRAM FOR FORHØJELSE AF TIDS-TÆLLER
****  LINIE 300 :    ZEIT=&A052
****  LINIE 320 :    ZEIT=&A052
****  LINIE 350 :    ZEIT=&A052
A052 7E      490  ZEIT     LD    A,(HL) ; GAMMEL TID
A053 C601    500            ADD   A,1 ; FORHØJES
A055 27      510            DAA   ; BYTES I BCD-FORMAT
A056 77      520            LD    (HL),A ; GEMMES
A057 B8      530            CP    B ; MAKS. NÅET ?
A058 D8      540            RET   C ; NEJ, SÅ FÆRDIG.
A059 97      550            SUB   A ; JA, SÅ FORTSÆT MED 0
A05A 77      560            LD    (HL),A
A05B 2B      570            DEC   HL ; NÆSTE GANG MINUTTER
                                  (TIMER)
A05C C9      580            RET
A05D         590
A05D         600                  ; RUTINE FOR UDSKRIFT AF EN
                                  BCD-byte
****  LINIE 390 :    BCBYOU=&A05D
****  LINIE 420 :    BCBYOU=&A05D
****  LINIE 450 :    BCBYOU=&A05D
A05D 3E30    610  BCBYOU   LD    A,&30 ; (&30 = ascii 0)
A05F ED6F    620            RLD   ; ROTER HØJESTE VÆRDI CIFFER
                                  I AKK
A061 CD0000  630            CALL  PRINT
A064 ED6F    640            RLD
A066 CD0000  650            CALL  PRINT ; GENKALD CIFFER NED
                                  LAVESTE
A069 ED6F    660            RLD   ; VÆRDI
```

```
A06B 23        670              INC   HL ; NÆSTE GANG MINUTTER
                                      (SEKUNDER)
A06C C9        680              RET
A06D           690
****  LINIE 410 :   PRINT=&A06D
****  LINIE 440 :   PRINT=&A06D
****  LINIE 630 :   PRINT=&A06D
****  LINIE 650 :   PRINT=&A06D
A06D E5        700  PRINT       PUSH  HL
A06E F5        710              PUSH  AF
A06F 2A0000    720              LD    HL,(POS)
A072 24        730              INC   H
A073 220000    740              LD    (POS),HL
A076 CDD3BD    750              CALL  WRITE
A079 F1        760              POP   AF
A07A E1        770              POP   HL
A07B C9        780              RET
A07C           790
****  LINIE 370 :   POS=&A07C
****  LINIE 720 :   POS=&A07C
****  LINIE 740 :   POS=&A07C
A07C           800  POS         DS    2
****  LINIE 190 :   COUNT=&A07E
****  LINIE 210 :   COUNT=&A07E
****  LINIE 260 :   COUNT=&A07E
A07E 2C01      810  COUNT       DW    300
****  LINIE 380 :   STUND=&A080
A080 00        820  STUND       DB    0
A081 00        830              DB    0
****  LINIE 280 :   SEC=&A082
A082 00        840  SEC         DB    0
```

```
PROGRAM: UR
START:   &A000
SLUT:    &A082
LÆNGDE:  0083
FEJL:    0
```

VARIABELTABEL:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| WRITE | BDD3 | INTCOU | 012C | POSITI | 4700 | DOPPUN | 003A |
| START | A00D | ANZEI | A035 | ZEIT | A052 | BCBYOU | A05D |
| PRINT | A06D | POS | A07C | COUNT | A07E | STUND | A080 |
| SEC | A082 | | | | | | |

```
 10 REM BASIC LOADER FOR UR
 20 REM INITIALISERING MED CALL &A000
 30 FOR I=&A000 TO &A07F
 40 READ A$:W=VAL("&H"+A$)
 50 S=S+W:POKE I,W:NEXT
 60 IF S<>14784 THEN PRINT"FEJL I DATALINIER":END
 70 ' FOR 464:IF S<>14776 THEN PRINT"FEJL I
    DATALINIER!":END
 80 PRINT"OK!":END
 90 DATA 21,0D,A0,22,51,B9,21,B1
100 DATA ' 464:..,..,..,..,49,..,..,..
110 DATA 00,22,01,A0,C9,21,B1,00
120 DATA E5,2A,7E,A0,2B,22,7E,A0
130 DATA 7C,B5,C0,21,2C,01,22,7E
140 DATA A0,21,82,A0,06,60,CD,52
150 DATA A0,3B,0A,CD,52,A0,38,05
160 DATA 06,24,CD,52,A0,21,00,47
170 DATA 22,7C,A0,21,80,A0,CD,5D
180 DATA A0,3E,3A,CD,6D,A0,CD,5D
190 DATA A0,3E,3A,CD,6D,A0,CD,5D
200 DATA A0,C9,7E,C6,01,27,77,B8
210 DATA D8,97,77,2B,C9,3E,30,ED
220 DATA 6F,CD,6D,A0,ED,6F,CD,6D
230 DATA A0,ED,6F,23,C9,E5,F5,2A
240 DATA 7C,A0,24,22,7C,A0,CD,D3
250 DATA BD,F1,E1,C9,00,00,2C,01
```

I tilfælde af at læseren ikke er i besiddelse af en Assembler, har vi lavet en BASIC-loader.

Der følger endnu et BASIC-program, hvormed uret kan indstilles. Ønsker man at standse uret, d.v.s. at fjerne det fra interrupt, så kan dette lade sig gøre, ved at kalde rutinen for anden gang med CALL &A000. Skal uret herefter reetableres, kan det først ske med CALL efter at følgende linie er indtastet:

POKE &A001,&D:POKE &A002,&A0

```
 10 REM URSTILLER
 20 MEMORY &9FFF
 30 MODE 2
 40 ' LOAD"UHR1.OBJ"
 50 LOCATE 15,7:PRINT"URSTILLER"
 60 LOCATE 1,11
 70 INPUT"12 ELLER 24 TIMERS VISNING (12/24) ?";E
 80 PRINT
```

```
 90 IF E<>12 AND E <>24 THEN 70
100 POKE &A031,VAL("&"+STR$(E))
110 ZEIBAS=&A080
120 INPUT"TID (HH,MM,SS);H,M,S
130 PRINT
140 POKE ZEIBAS,VAL("&"+STR$(H)):POKE ZEIBAS+1,VAL
    ("&"+STR$(M)):POKE ZEIBAS+2,VAL("&"+STR$(S))
150 INPUT"POSITION (KOLONNE,LINIE)";SP,ZE
160 IF SP<1 OR ZE<1 THEN 150
170 IF SP=1 THEN SP=257
180 POKE &A036,ZE-1:POKE &A037,SP-2
190 ' CALL &A000
```

# TREDIE DEL

# TIPS & TRICKS TIL MASKINSPROG

# 13. PROGRAMMERING I MASKINSPROG

Når hastighed (tid) går hen og bliver en afgørende faktor i programmering, er programmøren næsten altid tvunget til at ty til direkte eller indirekte programmering i maskinsprog. Med direkte menes der, at koden indtastes direkte/umiddelbart i computeren. Til dette formål findes der et utal af assemblere, der kan lette arbejdet. Den indirekte metode er den, der gives med en compiler. Har man f.eks. en BASIC-compiler, så skrives selve programmet på normal vis i BASIC. Bagefter oversætter compileren BASIC-programmet til maskinkode, der kan køre uafhængig af BASIC-fortolkeren.

For at få et indblik i denne verden, vil vi på baggrund af den direkte assemblering fremstille nogle rutiner, der skal vise den tidsmæssige fordel mellem et interpreterende (fortolkende) sprog og maskinkoden. Fortolkeren, der benyttes er BASIC'en , som findes på maskinerne CPC 464/664. Vi behøver dog ikke at anvende en assembler. I stedet vil vi bruge en BASIC-loader, der kan anbringe maskinkoden de rigtige steder i hukommelsen.

Prøv engang følgende lille BASIC-program:

```
 50 MODE 2
100 FOR ADRESSE=49152 TO 65535
110 POKE ADRESSE,255
120 NEXT
```

Programbeskrivelse:

De øverste 16K RAM i CPC-hukommelsen (&000-&FFFF) er reserveret til skærmbilledet. Det numeriske indhold i hukommelsescellerne repræsenterer bitmønsteret, der samlet på skærmen danner de velkendte tal og bogstaver (+ specialtegn).

Med vort eksempel "tænder" vi alle punkter på skærmen.

50 Kommandoen MODE 2 sletter skærmbilledet og sætter offset til øverste venstre hjørne.

100 FOR-NEXT-løkken begynder med første skærmadresse og slutter med den sidste.

110 Værdien 255 POKE's ind i alle adresserne.

Det tager ca. 40 sekunder at udføre dette program. Ved et antal på 16384 adresser, betyder det omkring 2,4 millisekunder pr. adresse. For at kunne sammenligne tiden med et tilsvarende program i maskinkode, kan vi prøve at køre det følgende program:

```
 10 MODE 2:SUM=0
 20 FOR I=40960 TO 40979
 30 READ A$:VERDI=VAL("&"+A$)
 40 POKE ADRESSE,VERDI:SUM=SUM+VERDI:NEXT
 50 IF SUM <> 1531 THEN PRINT"FEJL I DATALINIER":END
 60 PRINT "ET TRYK PÅ EN TAST STARTER PROGRAMMET"
 70 CALL &BB06:CALL 40960
 80 END
 90 DATA 21,00,40,01,01,00,11,00,C0,3E
100 DATA FF,12,13,ED,42,C2,0B,A0,C9,00
```

Denne assemblerrutine forkorter udførelsestiden med ca. 0.2 sekunder, hvilket pr. adresse er godt 12 mikrosekunder. Selvom denne rutine ikke viser den hurtigste måde at gøre det på, burde den alligevel give et billede af mulighederne med assemblering.

Programforklaring:

Maskinkodeprogrammet består af de hexadecimale værdier i DATA-linierne 90 og 100. I FOR-NEXT-løkken (linie 20-40) læses værdierne og konverteres til en numerisk (fra strengvariablen) værdi (linie 30) og POKE's ind i adresserne 40960-40979. Kommandoen CALL &BB06 i linie 70 kalder en af operativsystemets rutiner, der afsøger tastaturet indtil en tast trykkes ned. Først da udføres den næste kommando. Her følger CALL 40960, der kalder den maskinkoderutine, vi har lagt ned i hukommelsen med FOR-NEXT-løkken. Efter udførelse vender BASIC-fortolkeren tilbage (END i linie 80) med meddelelsen READY.

Men hvad betyder HEX-værdierne i DATA-linierne?

For at kunne give et tilfredsstillende svar på det, må vi kaste et par blikke på Z80 CPU'en, hjertet i Amstrad-computeren.

Lad os kigge lidt i kassen, med tricks til computeren. Læseren vil blive præsenteret for flere vigtige, hurtige og kraftige Z80-kommandoer, samt nogle praktiske eksempler på rutiner.

For at lette forståelsen, er det nødvendigt at kende til de forskellige registre. Hvordan de er bygget op, hvordan de fungerer og hvordan de bedst anvendes.

## 13.1. Z80's registre

Registrene er lageradresser i CPU'en. Selvom Z80 er en 8-bits processor, råder den også over 16-bits registre. Registrenes betegnelser lyder således:

8-bits registre: A, B, C, D, E, F, H, L

16-bits registre: BC, DE, HL, SP, PC

Hermed er de vigtigste registre blevet nævnt. De resterende vil vi springe over, da de ikke er nødvendige for den grundlæggende viden. Den opmærksomme læser har måske allerede fundet ud af, at nogle af 16-bits registrene har betegnelser, der kan dannes ud fra 8-bits registrenes navne. Der er en logisk grund til dette, men derom senere.....

## 13.1.1. 8-bits registre

De viste 8-bits registre har ikke ens værdier. A- og F-registrene har i forhold til de øvrige en højere position. Opdelingen burde måske se således ud:

A, F,  B,C,D,E,H,L

A-registeret (AKKumulatoren)

Alle logiske og aritmetiske 8-bits operationer foregår via A-registeret. Resultaterne af sådanne operationer står ligeledes til rådighed i AKK. De andre registres indhold ændres ikke herved.

1. Eksempel på to tals addition.

```
LD   A,6    Load AKK med tallet 6
LD   D,3    Load B-registeret med tallet 3
ADD  A,D    Adder indholdet af D med AKK
```

Efter operationen ADD A,D indeholder AKK værdien 9, imens D stadigvæk indeholder tallet 6.

2. Eksempel på to tals subtraktion.

```
LD   A,&54  Load AKK med &54
LD   L,&54  Load L-registeret med &54
SUB  A,L    Subtraher indholdet af D fra AKK
```

Efter subtraktionen indeholder AKKumulatoren værdien 0.

De ovenfor beskrevne aritmetiske operationer har ikke kun indflydelse på indholdet i AKK. Der sker også noget i det såkaldte Flag-register. Det giver forskellige udslag alt efter påvirkningen af operationer i AKK. For at kunne skelne de forskellige påvirkninger fra hinanden, bliver de enkelte bits i F-registeret enten sat eller slettet. Det er først på baggrund af de informationer, der kan læses ud af F-registeret, man kan foretage betingede operationer. Men nu først beskrivelsen af registeret.

F-registeret (Flag-register).

Som vi allerede har nævnt, er det indholdet af F-registeret, der er afgørende for tolkningen af de aritmetiske og logiske operationer. Zero-flag er kun eet af de seks flag, dette register råder over. Men sammen med Carry-bit (C), er det vel det vigtigste.

Flag-registerets opbygning:

I det følgende ses ordningen af de enkelte flag i Flag-registeret.

| BETEGNELSE | S | Z | | H | | P/V | N | C |
|---|---|---|---|---|---|---|---|---|
| BIT NR. | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Flagenes beskrivelse:

| | | | |
|---|---|---|---|
| S: | SIGN | 0 | Resultat positivt |
| | | 1 | Resultat negativt |
| Z: | ZERO | 0 | Resultat forskellig fra 0 |
| | | 1 | Resultatet er 0 |
| H: | Halfcarry | 0 | Ingen mente fra bit 3 til bit 4 i AKK |
| | | 1 | Mente fra bit 3 til bit 4 |
| P/V: | Parity/Overflow | 0 | Ulige paritet / ingen overløb fra bit 6 til 7. |
| | | 1 | Lige paritet / overløb fra bit 6 til bit 7 (yderligere funktioner følger senere). |
| N: | Subtraktions-flag | 0 | Efter udførelse af en addition. |
| | | 1 | Efter udførelse af en subtraktion. |
| C: | Carry | 0 | Ingen mente fra bit 7 til bit 8 |
| | | 1 | Mente fra bit 7 til 8. |

Fortegns-bit (S)

Med en byte's 8 bits kan der repræsenteres 2↑8=256 værdier, nemlig tallene 0 til 255. Ved fortegnsbehæftede tal er det værdier fra -128 til +127. Talværdien repræsenteres af bits 0 til 6 og fortegnet af bit 7. Er fortegnet positivt, slettes bit 7 (sættes til 0). Følgeligt, sættes bit 7 ved repræsentation af et negativt tal til 1. Det binære tal 11111111 svarer til det decimale -1.

Zero-bit (Z)

Hvis en byte berøres af en logisk eller aritmetisk operation, eller via en rotations- eller forskydningskommando, så har Z-flag indholdet 1, hvis byten er 0. Z-flag er 0, hvis byte-værdien er forskellig fra 0. Ved sammenligning mellem 2 bytes, f.eks. CP A,E bliver Z-bit 0, hvis registrene har samme indhold. Værdien 1 opstår, hvis indholdet er forskelligt i de to registre. Dette er nemt at forstå, hvis man ved at kommandoen CP A,S subtraherer værdien i S fra AKKumulatoren. Af de tre yderligere funktioner, Z-flag har, skal testkommandoen BIT b,r nævnes. Kommandoen BIT 5,D tester den 5. bit i register D for dets indhold. Er indholdet 1, så er Z-flag 0. Er det 0, er Z-flag 1.

Half-Carry-bit (H). H-flag viser mente (Carry) fra nedre nibble (bits 0-3) til øvre nibble (bits 4-7) i en byte. Den anvendes hovedsageligt til BCD-aritmetik og DAA (decimalsammenligning). Hvis den nedre nibble får en værdi højere end 9, skal der ske en menteoverførsel til den øvre nibble, da en nibble ved repræsentation af et decimaltal, ikke må være højere en 9. Den resulterede mente vises via H-flag.

Paritet/overflow-bit (P/V). Dette flag har flere forskellige funktioner.

a) Til trods for den høje standard, kan det ikke undgås, at der opstår fejl ved dataoverførsel, når det sker i så store mængder. Paritet anvendes mest ved overførsel af tegn i ASCII-format (7-bit), idet man hægter den 8. bit på som paritetsbit. Er der et lige antal 1'er bits, sættes paritetsbit til 1. Er antallet af 1'er bits ulige, sættes paritetsbit til 0. Sker der en fejl ved overførselen af data, så passer pariteten ikke. Det resulterer i, at der gøres et nyt forsøg på at overføre den samme byte.

b) Hvis resultatet af enten en addition eller en subtraktion bliver større end +127, så vil den 7. bit, der er reserveret fortegnet blive "misindformeret". Dette vises via P/V-flaget.

c) Ved Bloktransfer- og Bloksøgningskommandoer, vil dette flag blive aktualiseret via BC-register. Ved de nævnte kommandoer får den funktion som en tæller. Er tælleregisteret BC=0, sættes P/V-flaget til 0. Er BC forskellig fra 0, bliver P/V lig med 1.

d) Ved kommandoerne LD A,I og LD A,R vil P/V-flaget indeholde værdien af Interrupt-Enable-Flip-Flop'en IFF2. Dermed er det muligt henholdsvis at aflæse eller tilskrive IFF2.

Subtraktions-bit (N)

Programmøren vil ikke, ved normal programmering, få brug for dette flag. Den egentlige anvendelse for flaget er til decimalafrundingen (DAA), efter en addition eller en subtraktion. Afrundingen sker anderledes ved addition end ved subtraktion. Hvorledes CPU reagerer på DAA, ses i N-flag.

Carry-bit

Overløbs-bit'en viser efter udført addition eller subtraktion om der er sket overløb. Ved rotations- eller forskydningskommandoer anvendes carry-bit som 9. bit.

Carry reset'es af de logiske operationer AND, OR og XOR. Disse Z80-kommandoer anvendes hyppigt, når carry skal slettes.

Men hvorfor sletter logiske kommandoer C-flag?

Lad os kigge på følgende eksempel med værdien &D3 i AKK:

```
        11010011
AND     11010011
        --------
        11010011
        --------
```

Da der ved denne kommando, som ved OR og XOR, aldrig kan opstå overløb, heller ikke, hvis AKK sammenlignes med andre registre, vil CArry-Flag antage andre stillinger end 0. Disse kommandoer har altså "skjulte" evner, som Carry-slet-kommandoer!

LIDT OM FORVIRRINGERNE OMKRING, HVORNÅR HVILKE FLAG SKAL BRUGES, SLETTES ELLER SÆTTES.

Det kan ved første blik virke uforståeligt, hvordan f.eks. Zero-flag netop bliver 1, når det testede register antager værdien 0 ved en operation, og omvendt. Det viser sig at være helt selvfølgeligt, når man kigger på flagene som sandhedsværdier. Et flag kan antage to sandhedsværdier: sand og falsk. Et 1-tal står for SAND og et 0 for FALSK. Z80 råder over flere betingede hop-instruktioner. Om betingelsen for et hop er tilstede, afgøres ved flagenes stilling. Eksemplet med den hyppigt anvendte hop-kommando JP C,adresse, skal vise samspillet med flag-registeret. Ved addition mellem A-registeret og en anden værdi, blev resultatet en værdi højere end &FF. Som vi ved, er resultatet trods overflow, ikke blevet ødelagt, idet carry-bit tæller med som den 9. bit med værdien 2↑8 (=256).

Forekommer dette resultat, skal programmet forgrene sig til en anden programdel. Dette opnås med hop-kommandoen JP C,adresse, der netop aktiveres via C-bit (når denne er 1).

Hvis vi ønsker at udløse et hop, netop når C-bit er 0, skal vi gøre brug af kommandoen JP NC,adresse.

Registrene B, C, D, E, H, L

Disse registre har samme værdi. De kan tilskrives værdier direkte. Man kan loade dem med værdier fra andre registre, eller med deres egne, hvilket i mine øjne ikke har nogen betydning med mindre, man ønsker at "trække tiden ud". Registrene kan loades med RAM-indhold og indholdet kan returneres. Indholdet kan opereres med indholdet i AKK både logisk og aritmetisk, samt påvirkes med rotations- og forskydningskommandoer. Hermed er gennengået de vigtigste anvendelsesområder for registrene, anvendt som 8-bits registre.

## 13.1.2. 16-bits-registrene BC, DE, HL, PC, SP

8-bits-registrene B og C, D og E henholdsvis H og L kan sættes sammen, hvorved man er istand til at operere med specielle kommandoer i forbindelse med 16-bits-registre; og det i en 8-bits CPU!

Det har selvfølgeligt et helt klart formål. Hvordan skulle man f.eks. ellers kunne adressere den 40000. adresse iblandt 65536 mulige? Et 8-bits ord kan højst gabe over 2↑8=256 forskellige værdier eller adresser. Tilføjer man imidlertid 2 bytes til et 16-bits register, så har man mulighed for at adressere 2↑16=65536 adresser.

De tre 16-bit'er har dog ganske bestemte funktioner. Selvom de af struktur ikke er forskellige, kræver processoren, for at udføre bestemte operationer, de korrekte parametre i de rigtige registre.

HL-registeret kan f.eks. betragtes som en 16-bits-akkumulator. Der kan udføres tre forskellige aritmetiske operationer med DE eller BC via HL.

Register BC benyttes oftest som tæller. Det sker ved de såkaldte blok- og søgekommandoer, som vi kommer ind på senere.

PC-registeret (Program Counter).

For at udelukke midforståelser: PC-registeret er et rent 16-bits register. Man har kun en meget ringe indflydelse på det.

Et programs instruktioner står altid anført i et hukommelsesområde udenfor processoren. I selve processoren befinder sig kun den aktuelle kommando under udførelse. Denne kommando henter CPU'en fra netop den adresse, der befinder sig i PC. Program Counter kan altså betegnes som en slags pointer. Den peger hele tiden på den næste adresse, hvori den følgende kommando, eller dataværdi, befinder sig. Efter at processoren har hentet en kommando fra hukommelsen, flyttes PC til den nye aktuelle adresse. PC forhøjes med værdien, der svarer til længden af data (der findes jo 1-, 2-, 3- og 4-bytes-kommandoer).

SP-registeret (Stack Pointer).

Stack Pointeren har ligeledes en pointer-funktion. SP indeholder adressen på øverste STACK-element.

Men hvad er en STACK?

I al almindelighed, har man direkte adgang til enhver adresse i hukommelsen, men ved STACK-behandling, har man hele tiden kun adgang til det øverste element. Hvis man forestiller sig en kasse med bøger i, hvor man ønsker at læse bogen, som ligger nederst. For at komme til den, må man først fjerne alle de bøger, der ligger ovenover. Denne tilgangsmåde, kaldes LIFO-princippet. LIFO er en forkortelse for Last-In-First-Out (Sidst-inde-først-ude). Den sidst placerede bog i stabelen, er den første, som kan fjernes. Eksemplet med bog-stabelen har kun een fejl! En STACK i en computer bygges op oppefra og nedefter. Første emne i stakken ligger på det højeste sted. De næste elementer vil hele tiden få tildelt en lavere adresse. Det har dog ikke den store betydning, idet arbejdet, overtages af CPU'en. Man skal vide det, hvis man ønsker at flytte STACK til et andet område.

Det er dog ikke ret tit, man vil få brug for det, idet der i RAM altid er reserveret et område til STACK. Dette område må under ingen omstændigheder overskrives af andre data. Systemet vil uvilkårligt hænge sig op.

For fuldstændighedens skyld, skal det siges igen, at SP altid peger på det øverste, d.v.s. sidst placerede element i STACK.

Nu er de vigtigste registre i Z80-CPU'en gennemgået. Der findes andre registre, der dog ikke skal nævnes her, da de ikke er nødvendige for forståelsen.

Nu, hvor vi i det store hele er blevet kendt med Z80, kan vi gå igang med mikroprocessorens kommandoer. Da der ikke er nogen mening i, at lære dem udenad - der er over 500! - vil vi i starten kun nævne dem der kan hjælpe os med løsningen på forskellige problemstillinger.

## 13.3. Et detailleret eksempel på maskinsprogsprogrammering

Som det første programeksempel, vil jeg løse det, i starten af kapitlet, nævnte problem med sletning af skærmbilledet.

Det egentlige Assemblerprogram består, som vi ved, af værdierne i DATA-linierne 90 og 100.

```
 90 DATA 21,00,40,01,01,00,11,00,C0,3E
100 DATA FF,12,13,ED,42,C2,0B,A0,C9,00
```

Hovedformålet med BASIC-programmet var at få placeret disse værdier, i hukommelsesområdet fra adresse 40960 (&A000) og frem efter, via POKE-kommandoen.

DATA-linierne har kun visuel mening for ægte assembler-freaks. Men lad os lige prøve at se programmet på en forståelig måde. Denne måde kaldes for assemblerlistningens mnemotekniske (det er ikke en trykfejl!) form. Hvis man kan læse Hexkoden 3E FF og forstå, hvad den dækker over, så må man være i besiddelse af en særdeles god hukommelse (erfaring). Den mnemotekniske fremstilling lyder LD A,&FF. Heraf er det nemt, at se meningen:

| | | | | |
|---|---|---|---|---|
| LOAD | AKK | med | Hex-tallet | FF |
| LD | A | , | & | FF |

Den følgende programlistning er særdeles veldokumenteret. I første spalte er adressen angivet. I anden spalte står hex-kode, og i den tredie spalte står den mnemotekniske assemblerkode. Den sidste spalte indeholder bemærkninger til instruktionerne.

| Adresse | Hex-kode | Mnemonic | Bemærkning |
|---|---|---|---|
| A000 | 21 00 40 | LD HL,&4000 | Load tælleren |
| A003 | 01 01 00 | LD BC,&0001 | Load subtrahenten |
| A006 | 11 00 C0 | LD DE,&C000 | første skærmplads |
| A009 | 3E FF | LD A,&FF | Load AKK med FF |
| A00B | 12 | LD (DE),A | DE LOADes med AKK |
| A00C | 13 | INC DE | forhøj DE med 1 |
| A00D | ED42 | SBC hl,BC | Subtraher BC fra HL |
| A00F | C2 0B A0 | JP NZ,&A00B | Hop til &A00B hvis Zero-Flag=0 |
| A012 | C9 | RET | Retur til hovedprogram. |
| A013 | 00 | NOP | NOoPeration |

Adresserne i den venstre spalte svarer til den første byte i hver linie. Derfor springes hele tiden over 2 adresser efter 3-byte-kommandoerne. Det er noget, man hurtigt vænner sig til.

Listningens første 4 kommandoer, kender vi. De forskellige registre loades med værdier, der er til at forstå.

Nyt i listningen, er for det første kommandoen LD(DE),A. Når, som i dette tilfælde, et registernavn står anført i parentes, skal indholdet i registeret forstås som en adresseangivelse. DE indeholder altså på dette sted værdien &000 (se 3. linie). Derved flyttes indholdet, med kommandoen LD(DE),A , fra AKK til adresse &C000.

Den anden nye kommando er INC DE. INC står for INCrement, der betyder forhøjelse. Kommandoen forhøjer indholdet i det angivne register med 1. Efter denne kommando står nu værdien &C001 i register DE.

Iøvrigt, kan alle de nævnte registre INCrementeres. Ny er også kommandoen SBC HL,BC. SBC står for subtraktion med carry. Register HL formindskes med indholdet i BC. For at undgå vanskeligheder med denne kommando, skal man altid være opmærksom på, at en evt. sat carry-bit, kan gemme "sjove" overraskelser ved denne operation. I tvivlstilfælde m.h.t. denne bit's tilstand, skal man altid slette den. Det kan, som bekendt, gøres med kommandoerne AND A, XOR A og OR A. I dette programeksempel har jeg p.g.a. programmets gennemskuelighed undgået det.

Den sidste "nye" kommando i programmet (RETurn og NOP kan vel ikke siges at være ukendte), er den betingede hop-kommando JP NZ,&A00B. Den udføres altid, hvis betingelsen for hoppet er opfyldt. I hop-kommandoen JP NZ,adresse lyder betingelsen NZ (NonZero eller på dansk: Forskellig fra 0). Hvis resultatet af den foregående operation er lig 0, er betingelsen for et hop blevet opfyldt. I dette eksempel sker det så længe HL-registeret er 0, altså &4000 (16384 gange). Det svarer til antallet af adresser, reserveret skærmens hukommelse. Hvis kommandoen SBC HL,BC sætter register HL til 0, sættes ved samme lejlighed Zero-Flag. Betingelsen for hopkommandoens udførelse er ikke længere opfyldt. Næste kommando RET, flytter kommandoen tilbage til BASIC-fortolkeren. Den melder sig som altid med et READY.

Et kig på rutinen ud fra et tidsmæssigt skøn.

Hvis man til løsning af et programmeringsproblem, vælger at benytte maskinsprog for at opnå en højere udførelseshastighed, så må man også, for opbygningens skyld, overveje valget af kommandoer.

Ud fra den betragtning, er den netop viste rutine, faldet alt andet end heldigt ud. Dette skal dog demonstrere, for den opmærksomme læser, at den løsning, der ser elegant ud, ofte er alle andre løsninger, som ser omstændelige eller rodede ud, underlegen. Ved assemblerprogrammering gælder ikke ubetinget de samme regler, der gør programmering i højere sprog effektiv. Således er bl.a. sletning af et stort sammenhængende hukommelsesområde, såsom skærmbilledet, væsentlig hurtigere med et stort omfattende program, end med et lille program med få kommandoer. Som alle andre steder, skal der naturligvis være et rimeligt forhold imellem hastighed og programmeringsteknik.

For at kende hastigheden for udførelsen af en rutine, behøver vi ikke at stille os ved siden af computeren med et stopur. En lommeregner arbejder meget mere nøjagtig. Det fungerer naturligvis kun, hvis man ved, hvor længe CPU'en er om at udføre en enkelt kommando. I eksemplet med vores rutine, kender jeg svaret. Tallene efter mnemonics er en angivelse af kommandoens udførelsesvarighed i mikrosekunder.

```
        LD    HL,&4000      5
        LD    BC,&0001      5
        LD    DE,&C000      5
        LD    A,&FF         3.5
-------------------------------------
løkke   LD    (DE),A        6.5
        INC   DE            3
        SBC   HL,BC         7.5
        JP    NZ,løkke      5
-------------------------------------
        RET                 5
```

De fire første og den sidste linie i programmet udføres kun den ene gang, d.v.s. at de tilhørende tidsværdier kun skal tælles med een gang. Summen er 23.4*10↑-6 sekunder.

Programstykket, kendetegnet med hopmærket (LABEL) "løkke" og som er indrammet, skal gennemløbes &4000 gange. Derfor skal summen af tiderne for linierne multipliceres med &4000. Da summen er 22, tager sløjfens udførelse &4000*22↑-6 = 0.360448 sekunder. Lægger man tiden for den øvrige del af programmet til, får vi en total tid for programmets gennemløb, nemlig 0.3604715 sekunder.

At hop-kommandoen arbejder lidt hurtigere, hvis betingelsen er negativ, skal kun nævnes for fuldstændighedens skyld. I praksis betyder denne gevinst ikke noget.

Men lad os alligevel "tune" programmet lidt. Her skal vi gøre os klart, hvad der sker, når et registers indhold fortløbende forhøjes. På et eller andet tidspunkt, vil man uvilkårligt nå den maksimale værdi, der kan repræsenteres i et register. Overskrides værdien, vil registeret blive nulstillet (ligesom kilometertælleren i en bil), hvorefter der tælles opefter igen.

Dermed er vi nået til det punkt, hvor vi kan fortsætte med at gøre sletterutinen hurtigere. Hvis det omtalte register bliver 0, så har den forladt det, for os så interessante, område med skærmhukommelsen. Det skal være slutbetingelsen i vores rutine.

Indtil nu, har vi benyttet et selvstændigt tælleregister (HL=&4000) ved siden af adresseregisteret DE. Den ekstra tid, vi bruger til subtraktionen fra HL, vil vi spare i den nye rutine. Adresseregisteret DE nulstilles ved forhøjelse efter den sidste adresse &FFFF.

Efter at tælleren er blevet bortrationaliseret, ser programmet således ud:

```
        LD    DE,&C000    STARTADRESSE
        LD    A,&FF       FOR VÆRDI, DER SKAL GEMMES
løkke   LD    (DE),A      GEMMES.
        INC   DE          NY ADRESSE.
        JP    NZ,løkke    HOP VED BETINGELSE
        RET               TILBAGE TIL HOVEDPROGRAM.
```

Det synes at være en praktisk løsning. Der er sparet tre linier. De to dobbeltregistre HL og BC er fri til andet brug, og hvad mere vigtigt er, linien SBC HL,BC skal ikke mere gennemløbes. Det resulterer i en tidsbesparelse på 0.12288 sekunder, hvilket svarer til ca. 35 % af programtiden.

Desværre er der en stor hage derved, som alle nye assemblerprogrammører een eller flere gange vil erfare. Kommandoen INC DE har ikke indflydelse på flagregisteret, der skal antage sandhedsværdien, men har betydning for den betingede hop-kommando. Selv når DE forhøjes til 0, har det ingen indflydelse på Zero-flag eller de øvrige flag.

Følgelig vil betingelsen for retur-hoppet, aldrig blive opfyldt: Først belægges hele skærmområdet af &FF. Så forhøjes adresseregisteret DE til 0. Herved når man det nederste hukommelsesområde i computeren. Her ligger nogle for systemet livsvigtige rutiner, der nu overskrives. Computeren vil uundgåeligt hænge sig op.

Men forarbejdet er ikke spildt. Man må blot på en eller anden måde sørge for, at Zero-Flag sættes svarende til resultatet ved INC DE. Kommandoen BIT 7,D er løsningen. Hvordan den fungerer vil jeg beskrive efter listningen af det køreklare program:

```
        LD    DE,&C000
        LD    A,&FF
løkke   LD    (DE),A
        INC   DE
        BIT   7,D
        JP    NZ,løkke
        RET
```

Adresseregister DE bliver ved starten loaded med &C000. Af den binære notation ses de 16 bit's tilstande i dette tal:

| Register | D | | E | |
|---|---|---|---|---|
| Hexadresse | C | 0 | 0 | 0 |
| Binær | 1100 | 0000 | 0000 | 0000 |
| Bit nr. | 7654 | 3210 | 7654 | 3210 |

Som det ses, er bit 6 og 7 i register D fra starten sat, hvorimod de øvrige 14 bits er sat til 0. Ved forhøjelsen af DE, er det altså de 14 mindstbetydende bits, der ændres (de, der var sat til 0 fra starten). Først når DE får værdien &FFFF (11111111 11111111) og forhøjes endnu en gang, sker der overløb (overflow). Så er det, at de 16 positioner i registeret DE sættes til 0 (også bit 6 og 7 i D).

Det er dette øjeblik kommandoen BIT 7,D venter på og meddeler Zero-flag i modsætning til 16-bits-INCrement-kommandoerne. Kommandoen sætter Zero-flag til 1, når den testede bit i registeret antager værdien 0. I dette specielle tilfælde, kunne man også anvende kommandoen BIT 6,D, idet også den 6. bit i register D fra starten har værdien 1 og reset'es samtidigt med bit 7. Hvilken af de to bits, man vælger er ikke afgørende.

Tidssammenligningen mellem denne og den første rutine falder noget mere gunstig ud.

```
        LD    DE,&C000      5
        LD    A,&FF         3,5
løkke   LD    (DE),A        6,5
        INC   DE            3
        BIT   7,D           4
        JP    NZ,løkke      5
        RET                 5
```

Programmets gennemløbstid for løkken er på 18.5 mikrosekunder, hvor vi i den første rutine skulle bruge 22 mikrosekunder. Der er altså sket en klar forbedring, men vi er ikke tilfredse endnu.

I de indtil nu beskrevne rutiner, skulle løkkerne altid gennemløbes &4000 (16284) gange. Ved hvert gennemløb blev indholdet af AKK, d.v.s. en byte, altid lagt ud i skærmhukommelsen. Det skete med kommandoen LD (DE),A. Desværre findes der til Z80-processoren ikke en kommando lydende LD (DE),HL, hvor man ellers kunne belægge to adresser samtidigt. Det ville have betydet meget store tidsfordele. Derimod findes der kommandoer, der kan lagre indholdet af et 16-bits-register. Man kan ikke lave en elegant adressering, som det var tilfældet med kommandoen LD (DE),A, hvor det kun er register DE, der skal forhøjes.

For alligevel at kunne flytte to bytes med een kommando, så må man bruge et trick. Det hele går ud på at "misbruge" STACK's funktion. Normalt bruges STACK til f.eks. at mellemlagre indholdet af forskellige registre, hvis en anden rutine skal inddrages, hvor disse registre har andre formål. Yderligere, ligger STACK i et område i RAM, der ikke er tilgængeligt fra BASIC.

Trick'et går ud på, at flytte STACK til skærmhukommelsen for at kunne benytte den lynhurtige STACK-kommando PUSH rr. Kommandoen er med sine 6.5 mikrosekunder kun lidt hurtigere end en dobbeltudført LD (DE),A med 7 mikrosekunder, men den sørger i tilgift for at aktualisere adressetælleren, der skal medtages ved load-kommandoen. Dermed modsvarer PUSH-kommandoens 6.5 sekunder i virkeligheden den dobbelte LD (DE),A + dobbelt INC, ialt 13 mikrosekunder. Denne tidsgevinst behøver vist ikke flere kommentarer.

Kommandoen PUSH rr lagrer indholdet af registeret, der er angivet symbolsk med dobbelt r. Registrene, det drejer sig om er BC, DE, HL og AF (og IX,IY). Den nødvendige løkketæller, kan halveres til &2000, da antallet af gennemløb netop halveres med PUSH rr. I stedet for &4000*1 byte, lagres nu kun &2000*2 bytes.

Før vi går over til selve programmet, skal endnu en fatal fejlmulighed udelukkes. Stack Pointeren (SP) flyttes under programafvikling til skærmhukommelsen. For at den efter rutinen kan reetableres med sit tidligere indhold, skal man markere dette. Vi flytter indholdet til to adresser, der ikke påvirkes under afvikling af rutinen. De to adresser er &A030/31 og ligger i øverste halvdel af det område, BASIC kan adressere. I intervallet mellem vores programløkke, og inden RET-kommandoen, skal SP antage sin tidligere værdi. Glemmes dette, vil systemet uvilkårligt hænge sig op, hvilket iøvrigt er den bedste indikator for, at noget et gået galt!

Det skyldes at kommandoen CALL nn, der bruges til at kalde rutinen med, lægger sin returadresse og nogle parametre i STACK, hvorfra de hentes igen ved RET-kommandoens udførelse. Derfor skal SP kunne genfinde disse værdier.

Her er programmet:

```
        LD    (&A030),SP    Gem SP
        LD    SP,&FFFF      Stack til skærmhukommelse.
        LD    HL,&3FFE      Sæt tæller
        LD    BC,&FFFF      Værdi for lagring
løkke   PUSH  BC            Flyt til hukommelsen
        DEC   HL            Formindsk tæller
        BIT   5,H           Sæt flag
        JP    NZ,løkke      Betinget hop
        LD    SP,(&A030)    Aktualiser SP
        RET                 Tilbage til hovedprogram.
```

Løkken i dette program skal bruge 18.5 mikrosekunder, akkurat som den tidligere rutine. Men på samme tid lagres nu 2 bytes, hvor det før kun var een. Deraf ses det, hvor stærk en kommando PUSH er.

Nogen vil måske have lagt mærke til, at kommandoen LD (&A030),SP kun adresserer een lagercelle, hvorimod værdien fra SP, der skal flyttes, fylder to bytes. I adresse &A030 lægges den mindstbetydende byte og i adresse &A031, lægges den mestbetydende byte. Denne form for lagring af 2-bytes-værdier med først Low- og så High-byten benyttes generelt. *Det er altid Lo-byten, der lagres før Hi-byten.* Det er noget, man må huske, idet der ellers kan forekomme mange tilfælde, hvor man har svært ved at gennemskue, hvad der sker.

Ved kommandoen LD SP,(&A030), hvor dataretningen er modsat, overflyttes først Lo-bye, så Hi-byte. Et vigtigt eksempel: Hvis der skal ske et hop til adresse &6533, så hedder den tilsvarende kommando f.eks. CD 3365.

Til slut i dette kapitel vil jeg gennemgå en særdeles hurtig slette-rutine. Ved hjælp af logisk OR sættes flaget, der betinger hop-kommandoen.

```
        LD    (&A020),SP    Gem SP
        LD    SP,&FFFF      SP til skærmhukommelse.
        LD    BC,&FFFF      Load værdi
        LD    HL,&C001      Sidste adresse
        XOR   A             Slet AKK
        LD    (&C001),A     Slet sidste adresse.
løkke   PUSH  BC            Gem værdi
        OR    (HL)          Test for slut
        JP    Z,løkke       Betinget hop.
        LD    SP,(&A020)    Hent oprindelig SP
        RET                 Tilbage til hovedprogram.
```

Først hentes Stack-Pointerens værdi, der er særdeles vigtig for systemet, og flyttes til register &A020. Denne adresse ligger efter sletterutinen. Nu flyttes SP til skærmhukommelsen (LD SP,&(FFFF), register BC loades med værdien, der skal gemmes

(LD BC,&FFFF) og adressen på den sidste adresse til register HL (LD HL,&C001). Så loades AKK med 0, idet den er blevet XOR'et med sig selv, hvilket er en hurtig og elegant metode til dette. Da AKK allerede er 0, kan den bruges til at nulstille adresse &C001, d.v.s. lægge værdien 0 ned i den. Det er meget vigtigt, idet betingelsen for returneringen kun kan fungere som den skal, hvis denne adresse i starten har fået værdien 0 af AKK.

Hvis man på nuværende tidspunkt får forståelsesvanskeligheder, bør man prøve at foretage en simulering af løkke-gennemgangen på et stykke papir. Ellers kan det være vanskeligt at forstå instruktionen OR (HL). Hermed er det en selvfølge, hvorfor den tidligere nulstilling skulle finde sted.

I løkken testes hop-betingelsen via en sammenligning mellem indholdet i AKK og en adresse. Adressen adresseres indirekte via register HL.

Resultatet af OR-relationen er 0, sålænge både AKK og den omtalte adresse (C001) er 0. Så snart PUSH-kommandoen har flyttet værdien &FF til adresse &C001, er resultatet af OR-relationen forskellig fra 0. Dermed ender løkken.

Kigger vi på tiden igen, ved vi, at det der foregår udenfor løkken, ikke har nogen betydning. Vi står derfor med den følgende program-løkke:

```
løkke   PUSH BC          6.5
        XOR  A           3.5
        JP   NZ,løkke    5
```

Et gennemløb varer 15 mikrosekunder. Det betyder 0.246 sekunder for hele skærmhukommelsen eller 7.5 mikrosekund pr. byte.

Med dette sidste program, har vi nået den mest effektive løsning af problemet. Der skulle ikke kunne være nogen konstruktion, der arbejder hurtigere end med den ene lager-kommando (PUSH BC). Hvis antallet af PUSH-kommandoer stiger i løkken, vil den tidsmæssige andel af kommandoerne JP NZ og OR A tilsvarende formindskes. Ved 10 PUSH-kommandoer pr. løkke er det pr. byte 3.7 mikrosekunder, hvor det med een PUSH-kommando var 7.5 mikrosekunder. Ved hver ekstra PUSH-kommando nærmer vi os grænsen på 3.25 mikrosekunder, der svarer til halvdelen af en hel PUSH. Desværre nærmer vi os også, med den slags programering, hukommelsesgrænsen for computeren. Derfor er det en fornuftig løsning, vi har nået.

Nu er der sagt nok om løkken. Det skal dog nævnes igen, at SP ubetinget SKAL aktualiseres inden returneringen til hovedprogrammet. Herved sættes SP til sin oprindelige værdi, d.v.s. at den ved, hvor de for returneringen vigtige parametre befinder sig. En kommando, der ligner den, findes i BASIC, hvor vi har GOSUB-RETURN-kommandoen. Møder BASIC en GOSUB-kommando, så markerer den et sted, hvor kommandoen er mødt, således at den ved næste RETURN, ved hvor den skal hen. I Amstrad findes der en speciel BASIC-STACK til dette. Den arbejder ligeledes efter LIFO-princippet.

Ved lagring af indholdet i dobbeltregisteret BC, er vi naturligvis ikke tvunget til at tage værdien &FFFF. Der kan vælges enhver værdi mellem &01 og &FF, bare ikke 0. I det tilfælde, ville vi aldrig nå frem til retur-betingelsen. D.v.s. at vi har programmeret en fantastisk ophængningsrutine.

Den, der føler sig i form til det, kan forsøge at muliggøre det. Kommandofølgen er den samme. Der er kun et par værdier, der skal ændres. Ikke give op - inden de første store succeser indtræffer, vil computeren nok hænge sig op en del gange og det altid perfekte fejlfri program, vil vise sig at være en rigtig BUG.

Ønsker man at indbygge flere PUSH-kommandoer, så skal det siges med det samme, at en byte ikke må PUSH'es i hukommelsen direkte under &C000. Følgen vil være den bekendte mangel på kommunikation med computeren. Det skyldes, at hukommelsesområdet direkte under &C000, altså &BFFF, er basis for maskinstacken. Ved computerens opstart er det reserverede STACK-område mellem &BF00 og &BFFF.

For ikke at ramle ind i noget under &C000 under PUSH'ingen, kan man f.eks. give HL en højere værdi. Det virker ikke sandsynligt, vel? Altså må vi igang med en skrivebordstest (simulering på papiret). Det hjælper altid, når der opstår problemer. Der er selvfølgelig ingen grund til at gennemberegne hele skærmhukommelsen, men kun den del, der gør det muligt at gennemskue virkningen.

I den forbindelse, skal jeg anføre, at man ikke bør anse en sådan skrivebordstest for værende forbeholdt pattebørn og barnlige sjæle. Ved kodning og fejlsøgning, findes der ikke nogen bedre metode. Den er faktisk en selvfølgelighed blandt de professionelle programmører.

Her følger endnu engang den hurtige sletterutine, men nu i form af en BASIC-loader. Til den absolutte hop-kommando, har jeg indsat en relativ, der dog sinker løkken med ca. 1 mikrosekund, men det gør hele rutinen relokativ. Man kan således placere den, hvor man ønsker det i hukommelsen.

```
 5 MODE 2:SUM=0
10 FOR I=STARTVÆRDI TO STARTVÆRDI+28
20    READ WERT$:WERT=VAL("&"+WERT$)
30    POKE I,WERT:SUM=SUM+WERT:NEXT
50 IF SUM <> 3682 THEN PRINT"FEJL I DATALINIER"
60 DATA ED,73,20,A0,31,FF,FF,01,FF,FF,21,01,C0,AF
70 DATA 32,01,C0,C5,B6,28,FC,ED,7B,20,A0,C9,00,00
```

## 13.4. De stærke Z80-kommandoer

Z80-processoren råder over flere hundrede kommandoer. Det synes ved første blik at være en næsten uoverkommelig hindring, for at tilegne sig teknikken. Bare rolig! Ved nærmere betragtning, ses det, at en stor del af kommandoerne kan sammensættes i grupper, der praktisk talt besidder samme funktion og med forskellige koder for de enkelte registre.

Inden vi behandler nogle af de vigtigste kommandoer, vil jeg på dette sted forklare, hvordan jeg i det følgende vil vise indflydelsen på flag-registeret. Under forkortelsen for flagenes navne, er der anført enkelte tegn, der entydigt skal beskrive indflydelsen på det aktuelle flag.

| S | Z | H | P/V | N | C | |
|---|---|---|---|---|---|---|
| ? | * | 1 | ? | 0 | - | m/t/z |

| | |
|---|---|
| *: | Flag indflydes afhængigt af resultatet |
| -: | Flag uændret |
| 1: | Flag sættes (ubetinget) |
| 0: | Flag reset'es (ubetinget) |
| ?: | Flag ubestemt efter operation. |
| m: | Antal maskincycler. |
| t: | Antal taktcycler. |
| z: | Angivelse af nødvendig tid i mikrosekunder |
| b: | Nummer på en bit. |
| r: | Står for registrene A, B, C, D, E, H, L |

## 13.4.1. Enkelt-bit-kommandoer

Nu vil vi, godt rustet som vi er, gnnemgå ca. 1/3 af Z80-kommandoerne. Der findes f.eks. de 168 kommandoer, der kun berører de enkelte bits i registrene A, B, C, D, E, H og L. Det er kommandoerne:

BIT b,r, SET b,r og RES b,r

Kommandoen BIT b,r kender vi allerede, selvom det ikke var i denne form.

BIT b,r

Denne kommando tester bit'en angivet i parameteret i et angivet register.

Parameterne b og r står for bitnummer (b) og registernavn (r), hvor den mestbetydende bit i et register altid er nummer 7, og er venstrestillet, hvorimod den mindstbetydende bit altid er højrestillet og har nummeret 0. Forkortelsen r står for 8-bit-registernavnene.

| S | Z | H | P/V | N | C | |
|---|---|---|---|---|---|---|
| ? | * | 1 | ? | 0 | - | 2/8/4 |

SET b,r

Denne kommando sætter den aktuelle bit i det aktuelle register. Parametrenes betydning kender vi fra tidligere.

| S | Z | H | P/V | N | C | |
|---|---|---|---|---|---|---|
| - | - | - | - | - | - | 2/8/4 |

RES b,r

Kommandoen reset'er den angivne bit, d.v.s. til værdien 0.

| S | Z | H | P/V | N | C | |
|---|---|---|---|---|---|---|
| - | - | - | - | - | - | 2/8/4 |

Og hermed har vi faktisk gennemgået ca. 1/3 af alle kommandoerne! På lignende vis, er det muligt at gennemgå endnu 1/3. Det er slet ikke nødvendigt at kende alle kommandoerne fra starten. De tilegnes helt automatisk efterhånden, som man får brug for dem. Pludselig står man med et problem, der ikke kan løses med de kendte kommandoer. Man bladrer løs indtil den rigtige kommando dukker op; sådan! Skulle vi imidlertid lave en komplet fortegnelse her i denne bog, ville den fylde flere bind. Der findes iøvrigt masser af assembler-litteratur. Det er dog ikke nødvendigt, at man anskaffer sig speciallitteratur til gennemgangen af denne bog. Alle kommandoerne heri, bliver gennemgået udførligt.

Men inden vi går igang med de næste kommandoer, kommer alle enkelt-bit-kommandoerne. Vi mangler nemlig dette hold:

BIT b,(HL)  BIT b,(IX+d)  BIT b,(IY+d)

Den følgende kommando kræver lidt forklaring:

BIT b,(HL)

Register HL antager her adressen på hukommelsen (i dette tilfælde, altså ikke et register, men en adresse), hvis bit nummer b skal testes. Denne indirekte adressering kan spare en mængde frem- og tilbageloadning.

| S | Z | H | P/V | N | C | |
|---|---|---|---|---|---|---|
| ? | * | 1 | ? | 0 | - | 3/12/6 |

SET b,(HL) / RES b,(HL)

Disse kommandoer sætter bit nummer b i registeret adresseret via HL til 1 henholdsvis 0.

| S | Z | H | P/V | N | C | |
|---|---|---|---|---|---|---|
| - | - | - | - | - | - | 4/15/7.5 |

## 13.4.2. Kommandoer for rotation og forskydning

Alle rotations- og forskydningskommandoer (bortset fra den ene bekendte undtagelse) har følgende punkter fælles:

a) Hele indholdet i det aktuelle register forskydes mod venstre/højre. Efter en kommando, der har forskudt indholdet mod venstre, befinder det gamle indhold, fra bit 0, sig nu i bit 1. Det tidligere indhold i bit 1, er nu i bit 2 o.s.v.

b) Den sidste bit der flyttes, overføres til carry-bit, hvilket ikke er en nødløsning af pladsmæssige årsager, men tjener et godt formål. Ved bevægelsen til venstre er denne bit 7, ved bevægelse til højre, bit 0.

c) Det er logisk, at carry-flag altid berøres ved brug af disse kommandoer.

d) Alle rotations- og forskydningskommandoer kan anvendes med følgende operander, der repræsenteres ved s.

A B C D E H L (HL) (IX+D) (IY+D)

RL s

Indholdet betegnet via operanden, forskydes mod venstre. Bit 7 går til carry, hvis indhold flyttes til bit 0.

| S | Z | H | P/V | N | C |
|---|---|---|---|---|---|
| * | * | 0 | * | 0 | * |

RR s

Indholdet angivet via operanden, forskydes mod højre. Bit 0 flyttes til carry, hvis indhold flyttes til bit 7.

| S | Z | H | P/V | N | C |
|---|---|---|---|---|---|
| * | * | 0 | * | 0 | * |

RLC s

Indholdet af den, via operanden, angivne position, forskydes mod venstre. Derved flyttes bit 7 tilbage til 0 og tilsvarende i carry.

| S | Z | H | P/V | N | C |
|---|---|---|---|---|---|
| * | * | 0 | * | 0 | * |

RRC s

Indholdet, angivet via operanden, forskydes mod højre. Derved flyttes bit 0 til bit 7 og ligeledes i carry.

| S | Z | H | P/V | N | C |
|---|---|---|---|---|---|
| * | * | 0 | * | 0 | * |

SLA s

Denne kommando forskyder indholdet, der er angivet via operandens s, mod venstre. Den derved opståede frie bit 0 reset'es ubetinget. Bit 7 forskydes til carry.

| S | Z | H | P/V | N | C |
|---|---|---|---|---|---|
| * | * | 0 | * | 0 | * |

SRA s

Denne kommando forskyder indholdet angivet via operandens s mod højre. Bit 0 forskydes til carry. Bit 7 forbliver uændret.

| S | Z | H | P/V | N | C |
|---|---|---|---|---|---|
| * | * | 0 | * | 0 | * |

SRL s

Kommandoen forskyder indholdet, der er angivet via den af operanden betegnede position s, mod højre. Bit 0 forskydes i carry. Bit 7 reset'es ubetinget, d.v.s. sættes til 0.

| S | Z | H | P/V | N | C |
|---|---|---|---|---|---|
| * | * | 0 | * | 0 | * |

For de beskrevne rotations- og forskydningskommandoer gælder for de angivne operander "s", følgende tidstabel:

| s: | m | t | z |
|---|---|---|---|
| r | 2 | 8 | 4 |
| (HL) | 4 | 15 | 7,4 |
| (IX+d) | 6 | 23 | 11,5 |
| (IY+d) | 6 | 23 | 11,5 |

Rotations- og forskydningskommandoernes anvendelsesområder:

Efter at vi har lært nogle, ikke alle, R-S-kommandoer at kende, skal vi nu igang med nogle praktiske eksempler, hvormed kommandoernes nytte fremhæves.

Af de forskelligste grunde, er der sjældent mange muligheder for datatransmission. En ledning (kanal) kan naturligvis kun overføre een bit af gangen. Denne form for transmission (bit for bit) kaldes seriel overførsel. Det er ikke enkelthederne i den serielle datatransmission, vi her skal gennemgå, men kun belyse, hvorledes man deler en byte op i enkelte bits, som så kan overføres ad den serielle vej. Det forklares lettest med et kort eksempel.

Det forudsættes at byten, der skal overføres, befinder sig i register D.

```
        LD    B,8          Tæller 8 bits pr. byte.
løkke   XOR   A            Slet AKK
        SLA   D            Bit 7,D til carry
        JR    NC,M1        Spring over den følgende kommando.
        LD    A,1          AKK forskellig fra 0
M1      CALL  udlæsning    Hop til senderutine
        DEC   B            Ajourfør tæller
        JP    NZ,løkke     Næste bit
        RET                Tilbage til hovedrutine
```

Først tilskrives tælleren B med værdien 8., således at vi kan kontrollere antallet af bits. Så slettes AKK via XOR A. Med SLA D forskydes indholdet i D een position mod venstre, hvorved bit 7 flyttes til carry. Indeholdt den oprindelige 7. bit et 0, så er også carry 0 på dette tidspunkt. Så falder kommandoen JR NC,M1 positivt ud. Hoppet til m1 udføres. Således forbliver indholdet i AKK uforandret (0). Var bit 7 et 1 så ville carry-bit ligeledes være 1 på samme tidspunkt. Dermed ville JR NC,M1 falde negativ ud, og således ikke blive udført. I det følgende ville kommandoen LD A,1 sætte AKK til værdien 1.

Ved label M1, bliver der nu via CALL-udlæsning udført en udlæsningsrutine, der er indrettet på en sådan måde, at den sender et logisk 0, hvis AKK er lig med 0, eller et 1-tal, hvis AKK er forskellig fra 0. Hvordan dette sker i detaljer, og hvorledes logisk 0 og 1 ser ud, er ikke nødvendigt at vide.

Det er derimod vigtigt at vide, hvordan man via kommandoen SLA D berører Carry-bit på en sådan måde, at dens indhold kan styre en hop-kommando, og dermed kan sætte parametre for en senderutine.

Et andet udtryksfuldt eksempel på, hvordan man bruger SLA r, er den multiplikative fordobling af et register- eller en adresse.

Men først et eksempel på, hvordan man *ikke* skal programmere en sådan enkel multiplikation: 13*2

```
        LD    B,13        Multiplikant
        LD    C,2         Multiplikator
        XOR   A           Slet AKK
løkke   ADD   A,B         Add. multiplikation.
        DEC   C           Tæller
        JP    NZ,løkke    Hop ved C<>0
        RET
```

Først flyttes multiplikanten og multiplikatoren til henholdsvis register B og C. Så slettes AKK, hvori resultatet skal opadderes. I løkken, bliver AKK adderet med B register C gange.

Den følgende listning giver det samme resultat:

```
        LD    A,13        AKK=13
        SLA   A           Roter AKK mod venstre.
        RET
```

Denne multiplikationsmetode skal kort belyses ud fra den binære notation.

Register A  00001101 = 8+4+1=13  (Før)
Register A  00011010 = 16+8+2=26  (Efter)

Til højre for de binære tal, er summen af de binære positioner svarende til den binære positionering, af de satte bits. Forskydes den binære positionering mod venstre, så får hver position en højere værdi, nemlig altid den dobbelte. Når addenderne fordobles, så er den resulterende sum blevet fordoblet.

Spørgsmål: Hvad sker der, hvis tallet, der skal fordobles, er højere end 127?

Vi får en carry. Men for at kunne forvalte et sådant resultat, skal der bruges et nyt register, som kan tage sig af opdukkende carrys. På baggrund af det følgende eksempel, hvor tallet 160 skal multipliceres med 4 (160*4), vil sammenspillet mellem to registre blive forklaret. Tallet 160 placeres i register L, hvor det skal fordobles to gange.

```
        LD    H,0         Slet register H
        LD    L,160       " L = 160
        LD    B,2         Tæller
løkke   SLA   L           Forskyd L mod venstre
        SLA   H           Forskyd H mod venstre
        OR    A           Slet Carry
        DEC   B           Ajourfør tæller
        JP    NZ,løkke
        RET
```

Først slettes H. L loades med 160 og B sættes som tæller. I den efterfølgende løkke, er det altid register L, der forskydes først, idet dets indhold ved forskydning, vil sætte

carry. Denne skal derefter forskydes til H. Da man kan betragte registrene H og L som et 16-bits-register, har man efter gennemløb af rutinen det endelige resultat i HL.

Kommandoen OR A er kun taget med for en ordens skyld. Den skal slette en evt. carry ved H, for at forhindre at det forskydes til bit 0 i register L, ved næste gennemløb af løkken.

For at kaste lys over denne lidt uvante regningsmåde, har vi et eksempel i binære tal:

```
Register      H         L
              00000000  10100000 = 128+32=160 (før)
              00000001  01000000
              00000010  10000000 = 512+128=640 (efter)
```

Den, der har støvet hele Z80-assemblerkommandolisten igennem, vil have fundet ud af, at Z80 ikke har nogen 16-bits rotations- eller forskydningskommandoer.

Og alligevel, har den det. I de foregående eksempler simulerede vi en ikke tilstedeværende multiplikationskommando ved venstreforskydning af et register. Nu vender vi kagen og skaber en additionskommando via forskydning.

Som vi allerede ved, kan vi anse dobbeltregisteret HL for værende en pseudo-16-bits-akkumulator. Registrene BC, DE, SP og HL kan adderes med register HL. Da HL kan adderes med sig selv, hvilket jo svarer til multiplikation med faktor 2, må resultatet nødvendigvis svare til resultatet fundet via venstreforskydningskommandoen.

16-bits-kommandoen ADD HL,HL svarer altså i princippet til 8-bits-kommandoen SLA r. Indholdet af det angivne register forskydes mod venstre. Den mestbetydende bit flyttes til carry, og i den mindstbetydende bit forskydes et 0.

## 13.4.3. 16-bits aritmetik-kommandoer

ADD HL,rr

Indholdet, i det via rr betegnede dobbeltregister, adderes med indholdet af HL og flyttes til register HL. For rr, kan der blive tale om registrene BC, DE, HL og SP. Half-carry-bit'en flyttes via menteoverførsel fra bit 11 til bit 12, d.v.s. fra register H's mindstbetydende nibble til den højstbetydende. Der registreres ikke carry fra L til H.

| S | Z | H | P/V | N | C | |
|---|---|---|---|---|---|---|
| - | - | * | / | 0 | * | 3/11/5.5 |

Med denne kommando kan man, på nem vis, realisere en ikke i Z80 implementeret 16-bits rotationskommando.

Som hos en ægte AKK, kan man med register HL addere både med og uden carry. Kommandoen for addition med carry lyder:

ADC HL,rr

Indholdet af register HL og det via rr repræsenterede dobbeltregister adderes. Til denne sum adderes indholdet af carry-flag. For rr kan registrene BC, DE, HL og SP komme på tale. På nær N-flag, vil alle flag kunne berøres afhængigt af resultatet.

| S | Z | H | P/V | N | C | |
|---|---|---|---|---|---|---|
| * | * | * | * | 0 | * | 4/15/7.5 |

Med den følgende 16-bits-kommando kan alle HL-akkumulatorens aritmetiske evner beskrives.

SBC HL,rr

Indholdet af det via rr angivne dobbeltregister og mente subtraheres fra indholdet i register HL. Resultatet af denne subtraktion lægges tilbage i HL. rr dækker over følgende mulige registre: BC, DE, HL og SP. På nær N-flag kan alle flag påvirkes af resultatet.

| S | Z | H | P/V | N | C | |
|---|---|---|---|---|---|---|
| * | * | * | * | 1 | * | 4/15/7.5 |

## 13.4.4. Blok-kommandoer

Hvis man ønsker at flytte sammenhængende områder af RAM rundt i

hukommelsen, er der følgende Z80-kommandoer:

| | |
|---|---|
| LDI | Load med INC |
| LDIR | Load med INC med BRK |
| LDD | Load med DEC |
| LDDR | Load med DEC med BRK |

Hver af disse kommandoer består i princippet af flere Assemblerkommandoer udført i serie.

Bloktransfer-kommandoer.

LDI

LDI LD(DE),(HL); INC HL; INC DE; DEC BC

Indholdet af den med HL angivne adresse, loades til den adresse, der adresseres via DE. Derefter INCrementeres dobbeltregistrene HL og DE. Til slut DECrementeres BC.

| S | Z | H | P/V | N | C | |
|---|---|---|---|---|---|---|
| - | - | 0 | * | 0 | - | 4/16/8 |

LDIR

LDI LD(DE),(HL); INC HL; INC DE; DEC BC til BC=0

Indholdet af den med HL angivne adresse flyttes til adressen angivet via DE. Dobbeltregistrene HL og DE INCrementeres. Til slut DECrementeres BC. Denne kommandosekvens gentages, indtil register BC er DECrementeret til 0.

| S | Z | H | P/V | N | C | | |
|---|---|---|---|---|---|---|---|
| - | - | 0 | 0 | 0 | - | 4/16/8 | (BC=0) |
| | | | | | | 5/21/10.5 | (BC<>0) |

LDD

LDI LD(DE),(HL); DEC HL; DEC DE; DEC BC

Indholdet af adressen, angivet med HL, flyttes til adressen, der adresseres via DE. Derefter DECrementeres dobbeltregistrene HL og DE. Til slut DECrementeres BC.

| S | Z | H | P/V | N | C | |
|---|---|---|---|---|---|---|
| - | - | 0 | * | 0 | - | 4/16/8 |

LDDR

LDI (DE),(HL); DEC HL; DEC DE; DEC BC til BC=0

Indholdet af adressen, der angives i HL, flyttes til adressen, angivet i DE. Derefter DECrementeres dobbeltregistrene HL og DE. Til slut DECrementeres BC. Denne kommandosekvens gentages indtil register BC er DECrementeret til 0.

| S | Z | H | P/V | N | C | | |
|---|---|---|---|---|---|---|---|
| - | - | 0 | 0 | 0 | - | 4/16/8 | (BC=0) |
| | | | | | | 5/21/10.5 | (BC<>0) |

Her følger en kort sammenfatning:

LDI LD (DE),(HL); INC HL; INC DE; DEC BC
LDIR LD (DE),(HL); INC HL; INC DE; DEC BC indtil BC=0
LDD LD (DE),(HL); DEC HL; DEC DE; DEC BC
LDDR LD (DE),(HL); DEC HL; DEC DE; DEC BC indtil BC=0

Til alle CPC 464 eller 664 ejere, der med misundelse skæver til 6128'erens BANK-kommandoer, er her en rutine, der kopierer blokke af hukommelse.

```
LD    BC,&4000      Sæt tæller
LD    BE,&C000      Måladresse
LD    HL,&4000      Kildeadresse
LDIR                Blok-load-kommando
RET
```

Denne stærke rutine kan også indlæses med følgende BASIC-loader:

```
 5 MODE 2:SUM=0
10 FOR I=START TO START+11
20 READ WERT$:WERT=VAL("&"+WERT$)
30 POKE I,WERT:SUM=SUM+WERT:NEXT
40 IF SUM<>985 THEN PRINT "FEJL I DATA"
50 DATA 01,00,40,11,00,40,21,00,C0,ED,B0,C9
```

Efter rutinen er loaded med dette program, startes den op med kommandoen CALL start. Bortset fra READY-meddelelsen, sker der ikke noget. Rutinen har nemlig kopieret indholdet af hukommelsesblok 3, d.v.s. skærmbilledets hukommelse til blok 1 i RAM. Byt nu om på de understregede værdier i DATA-linien. Det er block-start-adressernes Hi-bytes. Efter en fornyet opstart af loaderen, bliver blok 1 kopieret tilbage til skærmbilledets hukommelse via CALL start. Husk at slette skærmbilledet med MODE 2 inden, for at opnå en bedre effekt.

Man er frit stillet m.h.t., hvorhen man kopierer skærmhukommelsen. Blot skal man passe på, ikke at kollidere med BASIC-programmer nedefter eller operativsystemet opefter. Længden af blokken er også fri. Jeg har kun benyttet skærmhukommelsen af praktiske grunde.

# 14. NOGLE MASKINKODERUTINER TIL SKÆRMBEHANDLING

## 14.1. Soft Scroll

Når computeren, under udlistning på skærmen, når frem til den nederste linie, rydder den plads til en ny linie, ved at forskyde hele skærmbilledet opefter, og slette den øverste linie. Det er ikke indholdet i hukommelsen, der forskydes, men forskydningen på skærmen opnås ved ændring af offset. Adressen, der er ansvarlig for indholdet i øverste venstre hjørne er således ikke længere &C000. Det er en anden hukommelse, der benyttes som skærmbasis. Med den følgende mini-loader kan man forsøge sig med denne form for scrollning:

```
10 FOR I%=&A000 TO &A005
20 READ WERT$:WERT=VAL("&"+WERT$)
30 POKE I%,WERT
40 NEXT
50 DATA 06,01,CD,4D,BC,C9
```

Skal skærmbilledet scrolle nedefter, byttes den markerede data-værdi ud med et 0. Vær opmærksom på, at skærmbilledet ikke allerede har scrollet, da der ellers, via den forskudte offset, kan opstå utilsigtede resultater.

I det følgende er en BASIC-loader, hvor scroll foregår ved at hver pixelrække (billedpunkter) forskydes enkeltvis. Indholdet scroller herved ovenud af skærmbilledet.

```
 10 MODE 2:SUM=0
 20 FOR I%=&A000 TO &A036
 30 READ WERT$:WERT=VAL("&"+WERT$)
 40 POKE I%,WERT:SUM=SUM+WERT
 50 NEXT
 60 IF SUM <> 5695 THEN PRINT"FEJL I DATA"
100 DATA 01,C7,C7,21,00,C0,22,00,B0,C5,2A,00,B0,54,5D,CD
110 DATA 26,BC,22,00,B0,01,50,00,ED,B0,C1,CD,09,BB,38,15
120 DATA 0D,20,E6,C5,11,80,FF,21,36,A0,01,50,00,ED,B0,C1
130 DATA 0E,C7,05,20,CE,C9,00,00,00
```

Skal skærmindholdet scrolles nedenud, skal linierne 100 og 110 skiftes ud som følger:

```
100 DATA 01,C8,C7,21,80,FF,22,00,B0,C5,2A,00,B0,54,5D,CD
110 DATA 29,BC,22,00,B0,01,50,00,ED,B0,C1,CD,09,BB,38,15
120 DATA 0D,20,E6,C5,11,00,C0,21,36,A0,01,50,00,ED,B0,C1
```

Testsummen er i dette tilfælde 5699.

Hvis ikke hele skærmbilledet skal scrolle, men kun en del i venstre side, så indtast de følgende kommandoer efter ovenstående:

```
POKE &A016,10 : POKE &A02B,10
```

10-tallet står for den scrollende del af skærmens bredde.

Her følger nu et "krympet" program, der er væsentlig hurtigere end det ovenstående, men som ikke kan afbrydes. Den stopper automatisk efter udscrollning af hele skærmen. Man kan benytte den ovenfor viste BASIC-loader. FOR-NEXT-løkken skal løbe fra &A000 til &A02F og testsummen er nu 4368. DATA-linierne ændres som vist herunder:

```
100 DATA 01,C8,C8,21,80,FF,22,00,B0,C5,2A,00,B0,54,5D,01
110 DATA 00,08,ED,42,CB,74,20,04,01,B0,3F,09,22,00,B0,01
120 DATA 50,00,ED,B0,C1,0D,20,E1,0E,C8,05,20,D6,C9,00,00
```

## 14.2. Sidelæns scrollning af nederste linie

Til fremhævet udlæsning af tekstelementer findes forskellige metoder. Der findes inversering, farvet, indramning, understregning og blinkende form o.s.v. En ny effekt er en slags lysavis, der fremhæver en tekst, så den er umulig at overse. Det følgende program får den nederste linie til at scrolle mod venstre. Teksten, der forsvinder ud over kanten vender tilbage i højre side.

```
A000  117702      ANF  LD    DE,&277       En gennemgang.
A003  D5                PUSH  DE
A004  114F08            LD    DE,&84F       Sum.-Anf. adresse.
A007  ED5341A0          LD    (&A041),DE    Gem sum.
A00B  01084F      M2    LD    BC,&4F08      Tæller-8-linie/&4F-sp.
A00E  21CEC7            LD    HL,&C7CE      Anf. adresse
A011  B7                OR    A             Slet Carry
A012  CB16              M1    RL    (HL)
A014  2B                DEC   HL            Ny adresse
A015  05                DEC   B             Linie slut?
A016  20FA              JR    NZ,M1
A018  114F00            LD    DE,&004F
A01B  3005              JR    NC,M3         Overtag Carry ?
A01D  19                ADD   HL,DE         Tidligere 1. Adr.
A01E  CBC6              SET   0,(HL)        Sæt højre bit.
A020  1803              JR    M4            Spring over RES
A022  19          M3    ADD   HL,DE         Tidligere 1. adr.
A023  CB86              RES   0,(HL)        Slet højre bit.
A025  ED52        M4    SBC   HL,DE
A027  064F              LD    B,&4F         Ny linietæller
A029  ED5B41A0          LD    DE,(&A041)    Hent summand
A02D  19                ADD   HL,DE
```

```
A02E 0D          DEC  C
A02F 20E1        JR   NZ,M1
A031 D1          POP  DE        Gem tællerværdi
A032 1B          DEC  DE
A033 CD09BB      CALL &BB09      Tast nede?
A036 3808        JR   C,ENDE    Break
A038 D5          PUSH DE
A039 CB7A        BIT  7,D       Aflæs tællerslut.
A03B 28CE        JR   Z,M2
A03D D1          POP  DE        STACK-korrektion
A03E 18C0        JR   ANF       Ny runde
A040 C9          ENDE RET
A041                            Adresse for summand
A042                            Adresse for summand
```

Skærmen må ikke have scrollet før kald af rutinen, da offset ellers er ændret. Programmet er beregnet for kørsel i MODE 2. I de to andre MODES opstår der uanvendelige, men interessante effekter.

Med de relative hop-kommandoer, er dette program i princippet relokativt. Det kan placeres overalt i den frie RAM. Men pas på med adresserne &A041 og &A042. I disse mellemlagres register DE. Skal programmet ligge et andet sted, bør også disse to adresser ændres tilsvarende.

BASIC-loaderen:

```
 10 MODE 2:SUM=0
 20 FOR I=&A000 TO &A042
 30 READ WERT$:WERT=VAL("&"+WERT$)
 40 POKE I,WERT:SUM=SUM+WERT
 50 NEXT I
 60 IF SUM <> 6591 THEN PRINT"FEJL I DATA"
 70 END
100 DATA 11,77,02,D5,11,4F,08,ED,53,41,A0,01,08,4F,21
110 DATA CE,C7,B7,CB,16,2B,05,20,FA,11,4F,00,30,05,19
120 DATA CB,C6,18,03,19,CB,86,ED,52,06,4F,ED,5B,41,A0
130 DATA 19,0D,20,E1,D1,1B,CD,09,BB,38,08,D5,CB,7A,28
140 DATA CE,D1,18,C0,C9,00,00,00
```

Efter at loaderen er startet op, ligger rutinen i hukommelsen og kan kaldes med CALL &A000 og standses med et tryk på en vilkårligt tast. Der er nu ikke længere brug for loaderen. Den slettes med DELETE 1-. Hvis man har beskyttet hukommelsesområdet med MEMORY &A000-1, kan man slette loaderen med NEW.

Eksempel på rutinens anvendelse:

```
10 LOCATE 5,25:PRINT"GENTAG INDTASTNINGEN"
20 CALL &A000
30 LOCATE 1,25:PRINT STRING$(80," ");
40 MODE 2
50 INPUT"NY INDTASTNING";A
60 REM ...ETC.....
```

## 14.3. Screen Copy til CPC 464/664

*Dette afsnit er først og fremmest tænkt anvendt af de læsere, som ejer en 464 eller en 664, og dermed ikke har de samme muligheder for skærmbehandling, som ejerne af 6128.*

Med SCREENCOPY kan man kopiere en blok i RAM, fra et sted til et andet. En blok er et sammenhængende stykke af hukommelsen med længden 16 KByte. Da 464/664 ialt råder over et 64 KByte stort RAM-område, svarer det til 4 blokke.

| | STARTADRESSE | SLUTADRESSE |
|---|---|---|
| BLOK 0 | &0000 | &3FFF |
| BLOK 1 | &4000 | &4FFF |
| BLOK 2 | &8000 | &BFFF |
| BLOK 3 | &C000 | &FFFF |

Selvom brugeren ved opstart af computeren har mere end 42000 bytes fri til programmering, kan man naturligvis kun bruge blok 1 helt frit.

Bemærkninger til hukommelsesblokkene: I blok 0 ligger de for operativsystemet nederste (ca 60) adresser lange "livsnødvendige" rutiner. Resten af denne blok er til fri udnyttelse. Men hvorfor "livsnødvendig"? Flyt værdien 0 til adresse 8 med POKE 8,0 - MEN KUN, HVIS DER IKKE ER NOGET VIGTIGT I COMPUTERENS HUKOMMELSE !!

Som allerede nævnt, er blok 1 helt til brugerens disposition.

I blok 2 kan adresserne &8000 - &A67B benyttes frit efter opstart. Området fra &A67C til &BFFF (decimalt 42620 - 49151) er ligeledes forbeholdt operativsystemet. Ændringer i dette område, skal som i det forrige afsnit, ske med forsigtighed og omtanke. *Husk at fjerne disketten, inden forsøgene starter!*

Blok 3 indeholder skærminformationerne. Ændrer man på noget her, så kan man direkte "aflæse" dette.

Hvorledes kommandoen SCREENCOPY fungerer, kan man se af eksemplet vist umiddelbart efter beskrivelsen af kommandoen LDIR. I dette eksempel blev skærmhukommelsen kopieret til blok 9. Hvis man først har kopieret et skærmbillede over i blok 1, kan det lynhurtigt erstatte det aktuelle skærmbillede. Denne effekt bruges iøvrigt i de fleste spilleprogrammer.

Forklaring:

Efter opstart eller efter RESET af computeren, henter hardwaren informationer for monitoren fra blok 3, altså fra det hukommelsesområde, der starter med adresse &C000. Hvis man alligevel ændrer startadressen, f.eks. til værdien &4000 (blok 1's 1. adresse), sammensættes skærmbilledet af data fra 16383 og &400 adresser frem.

Ligger der her informationer om et andet skærmbillede, er det dette billede, der vises. Det følgende program ændrer skærmbilledets basis:

```
10 FOR I=&A000 TO &A005
20 READ WERT$
30 WERT=VAL("&"+WERT$):POKE I,WERT
40 NEXT
50 DATA 3E,00,CD,08,BC,C9
60 INPUT"HVILKEN BLOK 1 / 3 ";B
70 IF B=1 THEN POKE &A001,&40 ELSE POKE &A001,&C0
80 CALL &A000
```

Den disassemblerede maskinkodelistning ser således ud:

```
A000  3E xx      LD    A,xx
A002  CD08 BC    CALL  BC08
A005  C9         RET
```

I første linie loades AKK med skærmadressens signifikante byte. I AKK overflyttes også parameter for rutinen SCR SET BASE, der kaldes med CALL &BC08. RETurn i sidste linie, behøver vel ikke længere nogen kommentarer. POKE-kommandoerne i BASIC-programmets linie 70 sørger for, at den valgte adresse for start af skærmbilledet, får POKEd sin Hi-byte i hukommelsen, således at AKK-load-kommandoen får en meningsfyldt værdi.

Der opstår en interessant variant for billedindholdet, hvis man efter start af BASIC-programmet, afbryder programmet med to gange ESC, MODE 2 og indtaster kommandoen CALL &BC08. På skærmen opstår der flere rækker punkter. De øverste viser de livsnødvendige systemrutiner og de resterende 4 rækker står for BASIC-programmet, hvormed skærm-startadressen blev ændret. Skærmområdet er nemlig flyttet til den første blok (blok 0), og man ser dette områdes programdele. Indtast et par BASIC-linier med linienumre. Læg mærke til ændringerne, når linierne overføres med ENTER-tasten.

# 14.4. SCREENSWAP for 464/664

Forskellen mellem SCREENCOPY og SCREENSWAP består i, at ved SWAP byttes indholdet i de to angivne områder. Ved COPY kopieres et område til et andet, hvorefter man har to identiske områder. Vi ved, at for ombytning af indholdet i to variabler, skal der bruges en tredie som mellemlager.

Eksempel: Byt indholdet af A og B (med x som mellemlager).

```
X=A : A=B : B=x
```

Ønsker vi nu at ombytte to hukommelsesområder, hvori der ligger billedinformationer, er der ikke noget fuldstændigt mellemlager til rådighed. Derfor bytter vi indholdet i små-bidder. Stykkerne er i det følgende eksempel af &100 bytes længde. Man kan selvfølgelig selv bestemme længden, så længe man ikke bruger for meget (husk at skærmbillede nr. 2 reducerer den frie RAM med 1/4). Der er heller ingen grund til at lave stykkerne for korte, idet rutinen så bliver for langsom.

BASIC-loader for SCREENSWAP på 464/664:

```
 5 START=&A000:SUM=0
10 FOR I%=START TO START+&40
20 READ WERT$:WERT=VAL("&"+WERT$)
30 POKE I%,WERT:SUM=SUM+WERT:NEXT
40 IF SUM <> 5098 THEN PRINT"FEJL I DATA"
50 DATA 21,00,C0,22,00,81,21,00,40,22,02,81,01
60 DATA 40,00,C5,01,00,01,11,00,80,2A,00,81,ED
70 DATA B0,01,00,01,ED,5B,00,81,2A,02,81,ED,B0
80 DATA 01,00,01,ED,5B,02,81,21,00,80,ED,B0,21
90 DATA 01,81,34,23,23,34,C1,0D,C5,20,D1,C1,C9
```

Assembler-listning til SCREENSWAP 464.664:

```
A000  21 00 C0        LD    HL,&C000
A003  22 00 81        LD    (&8100),HL
A006  21 00 40        LD    HL,&4000
A009  22 02 81        LD    (&8102),HL
A00C  01 40 00        LD    BC,&0040
A00F  C5              PUSH  BC
A010  01 00 01  løkke LD    BC,&0100
A013  11 00 80        LD    DE,&8000
A016  2A 00 81        LD    HL,(&8100)
A019  EDB0            LDIR
A01B  01 00 01        LD    BC,&0100
A01E  ED5B 00 81      LD    DE,(&8100)
A022  2A 02 81        LD    HL,(&8102)
A025  EDB0            LDIR
```

```
A027  01 00 01        LD    BC,&0100
A02A  ED5B 02 81      LD    DE,(&8102)
A02E  21 00 80        LD    HL,&8000
A031  EDB0            LDIR
A033  21 01 81        LD    HL,&8101
A036  34              INC   (HL)
A037  23              INC   HL
A038  23              INC   HL
A039  34              INC   (HL)
A03A  C1              POP   BC
A03B  0D              DEC   C
A03C  C5              PUSH  BC
A03D  20 D1           JR    NZ,løkke
A03F  C1              POP   BC
A040  C9              RET
```

Programmet er relokativt opbygget, men alligevel kan det ikke placeres hvor som helst i hukommelsen. Det må ikke ligge i blok 1, da denne bruges som skærm-mellemlager. SWAP'ningens mellemlager og fire reserverede adresser ligger i umiddelbar forlængelse af blok 1, i området &8000 - &8103. Dermed bortfalder området &4000 - &8103 for programmet.

Hvis man i BASIC-loaderens linie % ændrer START-adressen til &8104, opstår der et overskueligt område (&4000 - &8144), hvor der er rigelig plads til at SWAP'e på.

# 15. ET SIMPELT INTERFACE MELLEM BASIC OG Z80-REGISTRENE

Det er ikke nogen hemmelighed, at man af og til er bedre hjulpet med et par rutiner i maskinkode end de tilsvarende BASIC-kommandoer. Et godt eksempel er rutinen, der kaldes med CALL &BB06. Rutinen afventer det næste tryk på en tast. Når det sker, vender computeren tilbage til BASIC-programmet. &BB06 behøver ingen tilgangsparametre.

Kommandoen SPEED WRITE x gør at man via parametrene 0 og 1 kan vælge skrivehastighed for kassette. Hastigheden angivet med 1, er den højeste. Baud-hastigheden sættes til 2000. Men denne hastighed er selvfølgelig ikke den maksimale for CPC. Højere overførselshastigheder kan opnås med rutinen, der ligger fra &BC68. Der skal sættes 3 af Z80's registre. Det følgende program letter loadningen af registrene:

```
 10 MODE 2:SUM=0:GOSUB 100
 20 FOR I=1 TO 10
 30 READ REG$,PLATZ$:PRINT REG$;:INPUT" ";WERT$
 40 IF WERT$="" THEN WERT$="00"
 50 WERT=VAL("&"+WERT$):PLATZ=VAL("&"+PLATZ$)
 60 POKE &A000+PLATZ,WERT
 70 NEXT
 80 CALL &A000:END
100 FOR I=&A000 TO &A012:READ W$:W=VAL("&"+W$)
110 POKE I,W:SUM=SUM+W:NEXT
120 IF SUM <> 960 THEN PRINT"FEJL I DATA":END
130 RETURN
140 DATA 21,0,0,E5,F1,1,0,0,11,0,0,21,0,0,CD,0,0,C9,0
150 DATA A,2,F,1,B,7,C,6,D,A,E,9,H,D,L,C,HB,10,LB,F
```

Løkken i linierne 100 og 110 plus data i linie 140, danner det egentlige interface i form af en Assemblerrutine fra adresse &A000. Programmet spørger efter værdierne, der skal loades i registrene A, F, B, C, D, E, H og L. De følgende HB og LB betegner Hi- og Lo-byte for den valgte rutine. De indtastninger, der er uvæsentlige kan ignoreres med ENTER.

Dette enkle værktøj gør det nemt for os, at udnytte CPC's firmware fra BASIC. Men lad os vende tilbage til skrivehastigheden på kassettebånd. Tilgangsparametrene er længden for en halv null-bit i HL og fortestlængden i register A.

1000 BAUD: HL = 333 A = 25

2000 BAUD: HL = 167 A = 50

Værdierne giver en høj overførselssikkerhed. Hvor højt man ønsker at sætte hastigheden må, i sidste ende afhænge af båndkvaliteten og båndoptagerens tilstand (Er slette- og tonehovede blevet renset for nyligt ?) !!

# 16. AT PLACERE ET MASKINKODE-PROGRAM I HUKOMMELSEN

Den almindelige metode til at indlæse et maskinkodeprogram i hukommelsen, kender læseren sikkert. De fra DATA-linier læste værdier POKE's ind i den øverste halvdel af BASIC-RAM. Først er det benyttede hukommelsesområde blevet reserveret med MEMORY. Men metoden har en ulempe:

Da de egne symboldefinitioner også ligger i øverste halvdel af BASIC, sker der en overlapning. Systemprogrammørerne har kendt til dette problem, og har indbygget en spærreanordning. Er den øverste hukommelsesgrænse placeret under symboldefinitionerne med MEMORY, vil hver efterfølgende "SYMBOL-AFTER"-kommando resultere i fejlmeddelelsen "INPROPER ARGUMENT". Herved hindres det selvfølgelig ikke, at en udvidet symboldefinition overskriver et i hukommelsen liggende maskinkodeprogram. Det er på den ene side en god ting for maskinkodeprogrammer, men på den anden side resulterer hvert "SYMBOL AFTER" straks i en fejlmeddelelse. En mulighed for at hindre at programmet standser ved denne kommando, er at indtaste SYMBOL AFTER 0 i direkte mode.

Men for den sande programmør, er det ikke en tilfredsstillende løsning. En anden mulighed ville bestå i, at anbringe kommandoen i programmets første linie, og efter første gennemløb at slette linien (!).

For CPC 464 gælder følgende:

```
10 SYMBOL AFTER 100
20 A=PEEK(&AE81)+256*PEEK(&AE82)+1
30 POKE A+4,&C5
40 FOR I=A+5 TO A+PEEK(A)+256*PEEK(A+1)-2:POKE
   I,32:NEXT
50 MEMORY &A000
—
—
```

For CPC 664/6128 gælder:

```
10 SYMBOL AFTER 100
20 A=PEEK(&AE64)+256*PEEK(&AE65)+1
30 POKE A+4,&C5
40 FOR I=A+5 TO A+PEEK(A)+256*PEEK(A+1)-2:POKE
   I,32:NEXT
50 MEMORY &A000
—
—
```

En anden metode til variabel symboldefinering, der også tillader denne midt i et program, ændrer forbigående på de interne HIMEM-pointere, således at SYMBOL AFTER tillades. Benytter man den metode, skal man passe på at maskinkodeprogrammet ikke overskrives af symboldefinitioner.

For CPC 664 og CPC 6128 gælder:

```
10 MEMORY &9FFF
20 REM ....
30 OLDHIMEM=PEEK(&AE5E)+256*PEEK(&AE5F)
40 POKE &AE5E,PEEK(&AE60):POKE &AE5F,PEEK(&AE61)
50 POKE &B735,0
60 SYMBOL AFTER 200
70 MEMORY OLDHIMEM
80 REM...
```

For 464 gælder :

```
10 MEMORY &9FFF
20 REM...
30 OLDHIMEM=PEEK(&AE7B)+256*PEEK(&AE7C)
40 POKE &AE7B,PEEK(&AE7D):POKE &AE7C,PEEK(&AE7E)
50 POKE &B295,0
60 SYMBOL AFTER 200
70 MEMORY OLDHIMEM
80 REM...
```

Da de viste metoder kun er "medhjælpende", vil vi i det følgende gennemgå andre, som kan indsættes alt efter situationen eller behovet.

Først viser vi to muligheder, der i princippet udfører det samme, men i to forskellige afsnit af hukommelsen.

I stedet for at lagre et maskinkodeprogram i øverste halvdel af BASIC, er det naturligvis også muligt at lagre et program i den nederste halvdel.

Starten af BASIC bestemmes med en pointer i adresserne &AE81/2 (i 664/6128: &AE64/5). Normalt peger den på adresse &16F, d.v.s. at BASIC-programmerne starter ved adresse &170.

Flytter vi denne pointer til en større værdi, vi kalder LOMEM, har vi området fra &170 til LOMEM til rådighed for maskinkodeprogrammer. Altså:

```
POKE &AE64,INT (LOMEM/256)
POKE &AE65,LOMEM-INT(LOMEM/256)*256
NEW
```

(464: &AE69/5 => 6AE81/2)

NEW skal indtastes umiddelbart efter, da diverse pointere, bl.a. for variabeltabellen, ikke vil blive sat korrekt. Flytningen af området kan ikke mærkes ved normal programmering i BASIC. Selv symbol after fungerer, som den skal. Området i nederste halvdel af BASIC er således fremragende til at implementere maskinsprogsprogrammer i.

Med visse begrænsninger, er det muligt at lagre maskinsprogsprogrammer i System-RAM. Et eksempel på en sådan RAM, er KEY-memoryen. Startadressen ligger i &B4E1/2 (664/6128: &B62B/C). Normalt begynder tabellen ved adresse &B446 (664/6128: &B590); den er 152 bytes lang. Ulempen er at KEY-belægningen ikke længere er mulig. De udvidede taster kan reset'es med KEY DEF.

Det er også muligt at lagre tabellen med de af brugeren definerede tegn. Dog kan den overskrevne tegndefinition ikke bruges mere. Koden for det første tegn i tabellen står på adresse &B294 (664/6128: &B734). Størrelsen findes ved:

```
PRINT (256-PEEK(&B294))*8   (for 464)
PRINT (256-PEEK(&B734))*8   (for 664/6128)
```

Tegnkoderne 32-127 skal naturligvis forblive uændrede.

Tabellens startadresse står i adresse &B296/7 (664/6128: &B736/7), og slutadressen i adresse &B096/7 (664/6128: &b075/6). Det er også muligt at placere et maskinkodeprogram i Video-RAM (adresse &C000 til &FFFF). Dertil følgende overvejelse:

Video ram fylder 16 K (16K=16*1024=16384 bytes). I MODE 2 er der kun lagret 25*80*8 punkter. Forskellen, 384=8*48 bytes, bruges ikke.

Ved opstart, eller efter indtastning af MODE 2-kommandoen, ligger der 48 bytes ubenyttet hen, for hver adresse fra &C7D0-&CFD0, &CFD0-&CFFF, &D7D0-&D7FF...o.s.v. Problemet ved denne metode er, at skærmbilledet ikke under nogen omstændigheder må scrolle!

De frie områder vil ved scrollning forskydes og derved overskrive et evt. her residerende program. Alligevel skulle området nok kunne udnyttes, da der ellers ikke er nogen begrænsninger for dets anvendelse.

I det følgende vises endnu to metoder, der kun virker ved bestemte maskinkodeprogrammer. Alle koder, der lagres her, skal være relokative. D.v.s. at de ikke må indeholde nogen absolutte adresseringer, der kan ramme programmets eget adresseområde. Man kan heldigvis ofte skrive sit maskinkodeprogram, på en sådan måde, at der kun arbejdes med relative adresseringer/hop. Programmerne skal være af en sådan art, at de kan overtages af systemet og dets forvaltning. Man opnår herved, at programmet kan blive flyttet rundt efter behov, samtidig med, at hver byte i selve pogrammet er uændret.

Hvordan er det muligt?

Ganske elementært! BASIC tager sig løbende af forvaltningen af strenge (kædede bytes). En streng består af nogle efter hinanden følgende koder. Længden af en sådan "kæde" kan være op til 255 tegn. Derfor kan vi lagre et maskinkodeprogram, som een lang streng af bytes.

Koderne læses som sædvanligt fra DATA-linier og sættes derefter sammen een efter een til en streng med CHR$. F.eks.:

MAPRO$=MAPRO$+CHR$(byte)

Således er koden lagret. Den kaldes ved hjælp af @-funktionen, der peger på STRING-DESCRIPTOR (se programmørhjælp). Til vort program, der er indeholdt i MAPRO$, kan man skrive:

CALL PEEK(@MAPRO$+1)+256*PEEK(@MAPRO$+2)

Den anden mulighed bygger på det samme princip, men tillader lagring af maskinkodeprogrammer af vilkårlig længde.

Dennegang bruger vi en indekseret variabel af integer typen. Et integertal består af Hi- og Lo-byte. Koderne lagres efter hinanden i en tabel. En mere detaljeret gennemgang af indekserede variabler findes i afsnit 22. Det følgende program understreger princippet. Det er imidlertid vigtigt, at:

1) Tabellen skal være af integer-type (%).

2) Hvis programmet består af et ulige antal koder, skal &00 lægges til i sidste DATA-linie.

```
 10 LANG=26: " ANTAL BYTES I PROGRAMMET
 20 DIM KODE%(INT(LANG-1)/2))
 30 FOR I=0 TO INT((LANG-1)/2)
 40 READ L$,H$
 50 KODE%(I)=VAL("&"+H$+L$)
 60 NEXT I
 70 END
 80 DATA 3E,01,CD,0E,BC,CD,15,B9
 90 DATA 7C,21,57,86,FE,01,28,06
100 DATA 2E,5C,38,02,2E,77,CD,0B
110 DATA 00,C9
120 REM PROGRAM KALDES MED CALL @KODE%(0)
```

Efter at RUN er indtastet, kan programmet til enhver tid kaldes med CALL @KODE%(0). Det ved kald angivne indeks, skal være 0.

Er programmet prøvet? Læseren er måske blevet bange for, at computeren skal hænge sig op, eller at der vil ske RESET af systemet. Det eneste, der sker er at opstartsmeddelelsen kommer frem igen. Det lille program KODE% er interessant, idet det selv finder ud af, hvilken version det kører på. Det sker via systemrutinen KL Version Number.

Her følger programmets assembler-listning.

```
A000             10                    ; KODE
A000  3E01       20            LD      A,1
A002  CD0EBC     30            CALL    &BC0E ; SCR MODE A
A005  CD15B9     40            CALL    &B915 ; KL VERSION NUMBER
A008  7C         50            LD      A,H ; VERSION 0,1 ELLER 2
A009  215786     60            LD      HL,&8657 ; FOR 664
A00C  FE01       70            CP      1
A00E  28FE       80            JR      Z,OK
A010  2E5C       90            LD      1,&5C ; FOR 464
A012  38FE       100           JR      C,OK
A014  2E77       110           LD      1,&77 ; FOR 6128
****  LINIE 80 :       OK=&A016
****  LINIE 100 :      OK=&A016
A016  CD0B00     120   OK      CALL    &B
A019  C9         130           RET
```

```
PROGRAM:  KODE
START:    &A000
SLUT:     &A019
LÆNGDE:   001A
FEJL:     0
```

VARIABELTABEL:

OK A016

# 17. LAGRING AF ASSEMBLER-RUTINER OG HUKOMMELSES-OMRÅDER

Med CPC's fremragende BASIC-fortolker er man i stand til, at udvælge og lagre bestemte dele af RAM. Syntaksen for denne type SAVE-kommando ser således ud:

SAVE"programnavn",B,startadresse,længde

På den måde kan man gemme maskinkoderutiner på diskette eller bånd. F.eks.:

Programmet, man ønsker at gemme, ligger fra &A000 til &A013.

SAVE"DEMONAVN",B,&A000,&14

Pas på at længdeangivelsen ikke misforstås. Rutinens længde er &14 selvom sidste byte i adressen er &13. Mangler den sidste byte, oftest RET, crasher programmet efter opstart.

Man har også mulighed for at SAVE hele skærmbilleder.

SAVE"screen",B,&C000,&4000

I dette eksempel, er det det normale skærmområde, der begynder i adresse &C000 og slutter i adresse &4000, som gemmes i binær format. Nu er der en god grund til at overveje, at bruge de tidligere viste rutiner: SCREENCOPY og SCREENSWAP eller muligheden for at skifte skærmbasis. Da selve LOADningen af et skærmbillede tager nogle sekunder, kan man først anbringe billedet i RAM blok 1, og derefter kalde det lynhurtigt frem i skærmhukommelsen.

LOAD "SCREEN",&4000:CALL screencopy

# 18. NYTTIGE RUTINER I OPERATIVSYSTEMET

ADRESSE &0000: RST 0 - RESET

Kald af denne rutine med CALL 0, fungerer som tænd/sluk for computeren.

ADRESSE &0008: RST &08 - LOW JUMP

Denne rutine hopper til en adresse i operativsystemets ROM eller i den derover liggende RAM. Bit 14 og 15 bestemmer ROM/RAM-selektion. En sat bit betyder RAM, og en ikke sat bit betyder ROM. Bit 14 bestemmer over det nedre adresseområde (&0000 - &3FFF), imens bit 15 styrer det øvre område (&C000 - &FFFF).

ADRESSE &000C: JP (HL) MED ROM/RAM SELEKTION

Indirekte adresseret hop til adressen angivet i HL-registeret. Bit 14 og 15 varetager samme funktion som ved RST &08.

ADRESSE &0010: RST &10 - SIDE CALL

Tjener til kald af rutine i ekspansions-ROM.

ADRESSE &0018: RST &18 - FAR CALL

Tjener til kald af en rutine et vilkårligt sted i ROM eller RAM. Efter kommandoen RST &18 står adrssen på en vektor, der skal indeholde måladressen og ROM/RAM status.

ADRESSE &0020: RST &20 - RAM LAM LD A,(HL)

Akkumulatoren loades med værdien i den adresse, HL peger på. Via denne rutine selekteres RAM.

ADRESSE &0028: RST &28 - FIRM JUMP

Bruges til kald af en rutine i operativsystemet (firmware). Adressen angives umiddelbart efter kommandoen.

ADRESSE &0030: RST &30 - USER RESTART

Denne rutine er til rådighed for egne programmer. Et eksempel på RST &30's anvendelse findes i kapitel 23.

ADRESSE &0038: RST &38 - INTERRUPT TILGANG

Interruptrutinerne kaldes via denne adresse.

ADRESSE &BB06: KM (KEY MANAGER) - WAIT CHAR

ASCII-koden for en aktiveret tast lægges i AKK.

ADRESSE &BB15: KM EXP BUFFER

Rutinen fastlægger det RAM område, hvori belægningen af funktionstasterne lagres. Da området egentligt er ret lille, kan det gøres større med denne rutine. I DE-registeret angives startadressen. I HL-registeret angives bufferens længde. En maksimalt udvidet længde, afstedkommer funktionstaster hvortil en streng af længden 32 tegn kan defineres. Tidligere belægninger nulstilles ved brug af denne rutine.

ADRESSE &BB24: KM GET JOYSTICK

H-registeret indholder oplysning om joystick 1's stilling. L-registeret registrerer joystick 2's stilling.

ADRESSE &BB39: KM SET REPEAT

Repeat-funktionen bliver aktiveret når B<>0. Funktionen ophører når B=0.

ADRESSE &BB3C: KM GET REPEAT

Z-flag sættes, hvis tasten med det i AKK indeholdte nummer ikke er sat til repeat, ellers slettes flaget.

ADRESSE &BB3F: KM SET DELAY

Forsinkelse (tøven) i H-register, REPEAT-hastighed i L-register.

ADRESSE &BB42: KM SET DELAY

Overførsels tast-nummer: A, forsinkelsestid i H-register.
Returværdi: H: forsinkelsestid, L: repeat-hastighed.

ADRESSE &BB5A: TXT OUTPUT

Udskriver det, til værdien i AKK, svarende tegn på skærmen.

ADRESSE &BB60: TXT READ CHAR

Indlæser et tegn fra skærmen. I HL lægges tegnets position (H=linie / L=kolonne). Genkendes et validt tegn, sættes carry og tegnets ASCII-kode lægges i AKK.

ADRESSE &BB6C: TXT CLEAR WINDOW

Sletter det aktuelle skærmbillede.

ADRESSE &BB75: TXT SET CURSOR

H/L svarer til linie/kolonne.

ADRESSE &BB78: TXT GET CURSOR

ADRESSE &BB81: TXT CURSOR ON

ADRESSE &BB84: TXT CURSOR OFF

ADRESSE &BB90: TXT SET PEN

Pen=farven tilordnes det i AKK indeholdte INK-nummer.

ADRESSE &BB93: TXT GET PEN

Indlæser i AKK det aktuelle PEN-INK nummer.

ADRESSE &BB96: TXT SET PAPER (s. PEN)

ADRESSE &BB99: TXT GET PAPER (s. PEN)

ADRESSE &BB9C: TXT INVERSE

ADRESSE &BBA5: TXT GET MATRIX

Adressen på code A's tegndefinition lægges i register HL. Carry sættes, hvis det er et brugerdefineret tegn.

ADRESSE &BBA8: TXT SET MATRIX

Det af brugeren definerbare tegn med code A bliver ved hjælp af matrix'en loaded fra adresse HL.

ADRESSE &BBC0: GRA MOVE ABSOLUTE

DE-registeret indeholder X-koordinatet. HL-registeret indeholder Y-koordinatet.

ADRESSE &BBC3: GRA MOVE RELATIVE

DE-registeret indeholder det relative X-koordinat, og HL-registeret indeholder det relative Y-koordinat.

ADRESSE &BBC6: GRA ASK CURSOR

Registerbelægning som ved MOVE ABSOLUTE.

ADRESSE &BBC9: GRA SET PEN A=INK

ADRESSE &BBE1: GRA GET PEN A=INK

ADRESSE &BBE4: GRA SET PAPER A=INK

ADRESSE &BBE7: GRA GET PAPER A=INK

ADRESSE &BBEA: GRA PLOT ABSOLUTE

ADRESSE &BBED: GRA PLOT RELATIVE

DE: relativt X-koordinat.
HL: relativt Y-koordinat.

ADRESSE &BBF0 : GRA TEXT ABSOLUTE

DE: X-koordinat
HL: Y-koordinat
Resultat: A indeholder punktets INK

ADRESSE &BBF3: GRA TEXT RELATIVE

ADRESSE &BBF6: GRA DRAW LINE

Linien tegnes fra aktuel grafik-cursor og til målkoordinatet.

ADRESSE &BBF9: GRA DRAW LINE RELATIVE

ADRESSE &BBFC: GRA WRITE CHAR

Tegnet A udskrives på den angivne cursorposition.

ADRESSE &BC0B: SET LOCATION

Den aktuelle startadresse, for tegnet i øverste venstre hjørne, oplyses. A indeholder basisadressens Hi-byte (normalt &C0) og HL indeholder offset for staradressens basisadresse.

ADRESSE &BC0E: SCR MODE A

Den i AKK indeholdte skærm-mode aktiveres.

ADRESSE &BC11: SCR GET MODE

Det aktuelle MODE-nummer flyttes til AKK. Følgende flag påvirkes:

MODE 0 : C
MODE 1 : Z
MODE 2 : NC

ADRESSE &BC14: SCR CLEAR SCREEN MED INK 0

ADRESSE &BC1A: SCR CHAR POS

Overføres: H-linie / L-kolonne
Returneres: HL : 1. bytes adresse.
B: tegnets bredde i bytes (1,2 eller 3).

ADRESSE &BC1D: SCR DOT POS

Overføres: DE: X-koordinat / HL: Y-koordinat
Returneres: HL : Skærmadresse
C : Bit-maske, der kun angår det aktuelle punkt.
B : Antal punkter -1 pr. byte (1,2 eller 3)

ADRESSE &BC20: SCR NEXT BYTE

Skærmadressen i HL forhøjes 1 byte mod højre.

ADRESSE &BC23: SCR PREV BYTE

Skærmadressen i HL forskydes 1 byte mod venstre.

ADRESSE &BC26: SCR NEXT LINE

Sæt skærmadressen i HL til næste linie.

ADRESSE &BC29: SCR PREV LINE

Sæt skærmadressen i HL til tidligere linie.

ADRESSE &BC32: SET INK COLOR

A: INK-nummer, B og C farver.

ADRESSE &BC35: SCR GET INKCOLOR

ADRESSE &BC38: SCR SET BORDERCOLOR (B,C)

ADRESSE &BC3B: SCR GET BORDERCOLOR (B,C)

ADRESSE &BC3E: SCR SET FLASH TIME

H: tid for første farve.
L: tid for anden farve.

ADRESSE &BC41: SCR GET FLASH TIME

ADRESSE &BC5F: SCR LODRET LINIE

A: INK, DE : X-start, BC: X-slut, HL: Y-koordinat.

ADRESSE &BC62: CR LODRET LINIE

A: INK, DE: X-koordinat, BC: Y-slut, HL: Y-start.

ADRESSE &BC9B: CAS (h.h. DISK) CATALOG

ADRESSE &BCD1: KL (KERNAL) LOG EXT

Implementerer RSX-udvidelser.

ADRESSE &BD0D: KL TIME PLEASE

Flytter TIMER-værdien til DE og HL i 4-bytes format.

ADRESSE &BD10: KL TIME SET

ADRESSE &BD2B: MC PRINT CHAR

Udlæser værdien fra AKK til printer.

ADRESSE &BD2E: MC PRINTER BUSY

Tester om printer er RTR Ready To Receive. Hvis ikke, sættes Carry-flag.

ADRESSE &BD37: JUMP RESTORE : nødbremse

Flytter alle ændrede hop-vektorer tilbage til udgangspositioner.

ADRESSE &BDD3: WRITE

Indholdet i AKK udlæses som ASCII-tegn på skærmen.

# 19. NYTTIGE RUTINER I BASIC-FORTOLKEREN

Adressernes rækkefølge betyder:

Adresse 1 for 464, adresse 2 for 664 og adresse 3 for 6128.

ADRESSE &C356: &C3A0: &C3A3: PRINT

Udlæser det i AKK tilsvarende tegn på den aktuelle kanal.
Kanal-nummeret indeholdes i adresse &AC21, &AC06, &AC06.
Udskrivningen af vore programmer XREF/DUMP kan f.eks. skrives med POKE &AC06,8: |XREF (464: POKE &AC21,8|XREF).

ADRESSE &C34E: &C398: &C39B: LINE FEED

Udfører LINE-FEED på den aktuelle kanal (DEVICE)

ADRESSE &C43C: &C472: &C475: BREAK TEST

Tester om ESC er aktiveret. Hvis ja, hoppes til WAIT-mode (kendes fra BASIC).

ADRESSE &CA94: &CB55: &CB58: ERROR

Ved overførsel af fejlnummer til AKK (ved 664 og 6128 i E-register) udskrives den tilhørende fejlmeddelelse og programmet bringes til standsning (forceret break).

ADRESSE &D5DB: &D619: &D61C: Hent pointer for bogstav/variabeltabel.

ADRESSE &D6B3: &D6E6: &D6EF: Hent pointer for variabeltabel.

HL skal pege på starten af en variabel i programmet. Nærmere bestemt, på typekendetegnet. Eksisterer variabelen allerede i tabellen, returneres dens startadresse. Ellers inddrages variablen i listen og en ny startadresse returneres.

ADRESSE &DD37: &DE25: &DE2A: CHR NEXT

Tester om BASIC-programmets næste byte (adresse HL+1) svarer til byten, der følger kaldet af rutinen. Ved uoverensstemmelse udskrives en syntax error.

ADRESSE &DD3F: &DE2C: &DE31: CHRGET

Denne rutine sørger for at afhente den næste byte. Først forhøjes HL. Den i adresse HL beliggende byte læses. Er den BLANK (ASCII 32), læses den næste byte ... o.s.v.

Den første ikke-BLANK lægges i AKK. Er den en null-byte, sættes Z-flag. Rutinen bruges til step-vis læsning af et BASIC-program. En null-byte indikerer en linies afslutning.

ADRESSE &DD51: &DE3D: &DE42: CHRGOT

Læser endnu en gang den sidst læste byte og tester om slutningen på en kommando er læst. Hvis ja, sættes C.

ADRESSE &DD55: &DE41: &DE46: CHKKOMMA

Tester om den aktuelle byte repræsenterer koden for et komma. Hvis ja, læses den følgende byte og carry sættes. Ellers er carry-flag nulstillet.

ADRESSE &E8FF: &E9B9: &E9BE: BASIC DO RUTINE BC

Rutinen gennemløber det aktuelle BASIC-program og hopper efter læsning af starten på hver linie, til en vilkårlig rutine i adresse BC. Her kan linien undersøges. Efter behandling af linien, returneres med RET, hvorefter næste linie overtages.

ADRESSE &E943: &E9FD: &EA02: SKIP COMMAND

I samarbejde med sidstnævnte ignoreres hermed en linies kommandodel.

ADRESSE &EE79: &EF44: &EF49: PRINT HL decimalt

Rutinen kan benyttes til udskrivning af et BASIC-program's linienumre. Parameter er 2-byte-værdi i HL, repræsenterende linienummeret.

ADRESSE &F236: &F2D5: &F2DA: PRINT FAC

Denne rutine udlæser den for tiden indeholdte værdi i FAC i decimal form.

ADRESSE &FF4B: &FF6C: &FF6C: VAR FAC

Kopierer indholdet af en variabel i FAC. HL skal indeholde værdiens adresse.

ADRESSE &FF71: &FF92: &FF92: TEST bogstav

Indholdet i AKK bliver fortolket som ASCII og, hvis muligt konverteret til uppercase-format. Er koden ikke et bogstav i ASCII-format reset'es carry-flag.

ADRESSE &CFEE: &D058: &D055: MISSING OPERAND

ADRESSE &C205: &CB30: &C210: IMPROPER ARGUMENT

# 20. KOMPATIBILITET MELLEM DE TRE CPC'ER

I dette kapitel vil vi kort beskrive, hvor de vigtigste ligheder mellem de 3 Amstrad-computere CPC 464, CPC 664 og CPC 6128 hvad angår maskinsprogsprogrammering, ligger.

De 3 computeres ROM er i det store hele ens. I det store hele betyder desværre, at selv om alle rutinerne findes i samme form på computerne, så er tilgangsadresserne højst forskellige. Undtaget er heldigvis de fleste hop-vektorer i den centrale RAM. Helt præcist, er det vektorene for:

HIGH KERNAL - adresserne &B900 til &B920

KEY MANAGER - adresserne &BB00 til &BB4D

TEXT-pack - adresserne &BB4E til &BBB9

GRAfic-pack - adresserne &BBBA til &BBFE

SCReen-pack - adresserne &BBFF til &BC64

CAS- & DISK - adresserne &BC65 til &BCA6

SOUND-pack - adresserne &BCA7 til &BCC5

KERNAL-low - adresserne &BCC8 til &BD12

MaChine-pack - adresserne &BD13 til &BD35

JUMP RESTORE-vektor &BD37

Desværre, har vi ikke kunnet afprøve alle rutinerne. Vi har dog ikke i vores omgang med computerne opdaget, at nogen af disse rutiner ikke skulle have samme funktioner.

Efter JUMP RESTORE-vektoren i &BD37, er det dog slut med kompatibiliteten.

I 464 følger nu kald af LINE-EDITOR, og så hoppet til aritmetik-rutinerne. Ved 664 og 6128 er der nogle nye hopvektorer, til f.eks. FILL-rutinen. LINE-EDITOR-vektoren står i adresse &BD3A, og først herefter følger aritmetikrutinerne. Hop-vektoren (vektorer, der kun må benyttes ved indkoblet operativsystem), fra adresse &BDCD til adresse &BDF3 er derimod ens for alle computerversionerne.

Ønsker man at hoppe direkte til en rutine, som f.eks. alle BASIC-rutinerne, så er reglerne forskellige for de tre computere. Kun mellem CPC 664 og CPC 6128 er der ringe forskel.

Endnu en vigtig forskel er, at der ved CPC 664 og 6128 ikke længere findes CPC 464's indirections for BASIC-fortolkeren. De må være bortfaldet af pladshensyn. Det er af samme grund, at alle systemdataadresser (PEEKs og POKEs) er forskudte i forhold til 464. Disse adresser er dog i det store hele ens for 664 og 6128.

Konklusion:

Ved programmering, bør man holde sig til de nævnte vektorer. Er det ikke tilstrækkeligt, kan man nok ikke komme udenom, at skulle skrive 3 versioner af samme rutine. Det eneste lyspunkt er rutinen KL VERSION, der heldigvis (ellers ville alt være tabt), står på den samme adresse i alle versioner. (&B915). Denne rutine angiver nummeret på den aktuelle High-ROM. Herved indeholder H ROM-nummeret (1 for BASIC-ROM) og L versionsnummeret , hvor 0=CPC 464, 1=CPC 664 og 2=CPC 6128.

Se iøvrigt rutinen kaldet KODE i kapitlet om indlæsning af maskinkoderutiner.

# 21. KOMMANDOUDVIDELSE MED RSX

Det er forskellige muligheder for at udvide BASIC's kommandoforråd. På 464 kan man via PATCH-området i RAM udvide de forhåndenværende ROM-rutiner. Denne mulighed er dog temmelig begrænset, da man højst kan arbejde med 9 sådanne PACTH'es. Ved CPC 6128 er de af pladshensyn udeladt. Der findes dog en standardiseret metode for alle 3 computere. Metoden er kendt af de fleste. Den anvendes for alle AMSDOS-kommandoer.

På et eller andet tidspunkt, har de fleste vel prøvet at kalde CP/M-mode med !CPM, i hvert fald, de, der har en diskettestation. CPM-kommandoen starter med en streng (!). Ved indtastning af CPM (uden streg), opnår man kun en SYNTAX ERROR. Gør man det uden, at der er tilsluttet en diskettestation, får man UNKNOWN COMMAND.

Det drejer sig altså om en kommando, der kun kan bruges, når der er tilkoblet en diskettestation. Dermed er det en udvidelse.

"!" meddeler systemet, at det drejer sig om en udvidet kommando. Denne form for kommando er standard for alle Amstrad-versionerne. Metoden med at implementere kommandoer på denne måde, kaldes RSX. RSX står for Resident System Extension. Dette betyder at RSX er beregnet for alle systemudvidelser. Til disk-brug findes der en sådan udvidet system-ROM, der også indeholder kommandoerne !CPM og !ERA..o.s.v.

Vi kan imidlertid også gøre brug af RSX-kommandoen for at implementere vore egne kommandoer.

## 21.1. DOKE - Skrivning af en 2-byte-værdi i hukommelsen

Vi vil nu beskrive implementeringen med RSX-metoden via eksemplet med DOKE-kommandoen.

DOKE svarer til en "dobbelt POKE-kommando". Med POKE kan man tilskrive en adresse en værdi mellem 0 og 255. DOKE tilskriver to på hinanden følgende adresser med en værdi mellem 0 og 65500. Værdien placeres som Hi- og Lo-byte. Lo-byten lægges i den nederste adresse, og Hi-byten i den øverste. Med DOKE lettes ændringen af mange systemparametre i RAM (normalt via PEEK og POKE).

Parametertilskrivningen med RSX foregår analogt med CALL-kommandoen.

DOKE-kommandoen kræver 2 parametre. Adressen, hvorfra der skal lagres, og værdien der skal lagres. Kommandoen har følgende syntax:

!DOKE, adresse,værdi

Parametrene overleveres på følgende måde:

A: indeholder antallet af parametre.

Flag: Zero-flag=0, hvis ingen parametre er overgivet, ellers 0.

B: Indeholder 32-antallet af parametre.

DE: Indeholder den sidste overførte parameter.

IX: Adressen for det sidste element. Den næstsidste parameter står i adresse IX+2. Den tredie i IX+4..o.s.v.

DOKE-rutinen ser derefter således ud:

```
100 ' LD L,(IX+2)  ; OVERFØRT ADRESSE
110 ' LD H,(IX+3)  ; LOADES I REGISTER HL
120 ' LD (HL),E    ; GEM LO-BYTE
130 ' INC HL       ; SÆT ADRESSE FOR HI-BYTE
140 ' LD (HL),D    ; GEM HI-BYTE
150 ' RET          ; SLUT
```

Med CALL &A000,adresse,værdi kan vi nu kalde DOKE-rutinen, forudsat at den er blevet lagret fra &A000. Prøv engang:

CALL &A000,..,..

Dermed har vi en køreklar DOKE-rutine. Kaldet med CALL er lidt knudret. DOKE skal nu kaldes med DOKE. Derfor skal vi skrive en lille implementeringsrutine. Hovedarbejdet klares af rutinen "LOGEXT" i operativsystemet. Vores opgave bliver bare at give LOGEXT en værdi at arbejde med.

For at implementere DOKE, skal systemet vide følgende:

1) Udvidelsens navn.
2) Rutinens adresse.
3) Adressen på 4 ubenyttede bytes, der internt skal bruges til forvaltning af rutinen.

Adresserne på de 4 systembytes bliver, før kald af LOGEXT, loaded i HL-register. Yderligere loades BC med en startadresse for en tabel, hvori der gemmes yderligere informationer. Denne tabel indeholder, først og fremmest, adressen på en anden tabel. Nemlig den, i hvilken navnet eller navnene på flere implementerede rutiner indeholdes. I den anden tabels adresse i den 1. følger en hop-kommando til den

implementerede rutine(r). Implementeres flere rutiner, skal hop-kommandoernes og rutinernes orden følges.

Den anden tabel indeholder som nævnt, navnene på de enkelte rutiner. For at adskille navnene fra hinanden, skal bit 7 i det sidste bogstav i navnet altid sættes.

Men kig et øjeblik på vort eksempel:

```
 10 ' LD BC,RSXTAB ; ADRESSE PÅ DEN 1. TABEL
 20 ' LD HL,SYSBYT ; ADRESSE PÅ SYSTEMBYTEN
 30 ' CALL &BCD1 ; KALD AF RUTINEN LOGEXT
 40 ' RET ; IMPLEMENTERING SLUT
 50 ' RSXTAB DW NAMTAB ; TABEL 1 INDEHOLDER DEN 1.
    ADRESSE PÅ 2. TABELS STARTADRESSE.
 60 ' JP START ; OG SÅ HOP-KOMMANDO TIL RUTINEN
 70 ' NAMTAB DM "DOK"; NAVNETABEL
 80 ' DB &C5 : =ASC("E")+128 SÅ BIT 7 SÆTTES
 90 ' DB 0 ; NULL-BYTE KENDETEGNER SLUTNING PÅ
    NAVNETABEL
 95 ' SYSBYT DS 4 ; RESERVERING AF 4 BYTES FOR KERNAL
100 ' START LD.....HER BEGYNDER EN RUTINE
```

Efter assemblering af hele programmet (henholdsvis BASIC-loader) implementeres DOKE-kommandoen med CALL &A000. Fra dette tidspunkt er DOKE en fast kommandoudvidelse (RSX), d.v.s. at kommandoen kan gives både i direkte mode og i et program.

Pas på at implementeringsrutinen, kun kaldes den ene gang. De 4 systembytes bruges som pointere til evt. andre RSX-tabeller. Initialiseres den samme rutine for anden gang, så bliver pointeren, der peger på den næste RSX-tabel, sat til den samme tabel. Derved kan det ske, at kommandogangen kommer til at fortsætte i en uendelig løkke. Kommandoerne bliver ved med at kalde sig selv. Det følgende program forhindrer, at der kaldes igen, idet den tilføjer en RET-kommando i starten af initialiseringsrutinen.

```
A000  010000   10   INIT      LD    BC,RSXTAB
A003  210000   20              LD    HL,SYSBYT
A006  CDD1BC   30              CALL  &BCD1
A009  3EC9     40              LD    A,&C9
A00B  3200A0   50              LD    (INIT),A
A00E  C9       60              RET
****  LINIE 10 :    RSXTAB=&A00F
A00F  0000     70   RSXTAB    DW    NAMTAB
A011  C30000   80              JP    START
****  LINIE 70 :    NAMTAB=&A014
A014  444F4B   90   NAMTAB    DM    "DOK"
A017  C5       100             DB    &C5
```

```
A018  00        110             DB    0
****  LINIE 20 :    SYSBYT=&A019
A019              120 SYSBYT    DS    4
****  LINIE 80 :    START=&A01D
A01D  DD6E02      130 START     LD    1,(IX+2)
A020  DD6603      140           LD    H,(IX+3)
A023  73          150           LD    (HL),E
A024  23          160           INC   HL
A025  72          170           LD    (HL),D
A026  C9          180           RET
```

```
PROGRAM: DOKE
START:   &A000
SLUT:    &A026
LÆNGDE:  0027
FEJL:    0
```

---

VARIABELTABEL:

---

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| INIT | A000 | RSXTAB | A00F | NAMTAB | A014 | SYSBYT | A019 |
| START | A01D | | | | | | |

---

## 21.2. RPEEK - Random-Access-læsning fra RAM eller ROM

Den følgende listning viser, hvorledes man bruger @-funktionen i forbindelse med RSX eller CALL. I vort tilfælde bliver en integer-variabels værdis startadresse fundet med @, og derefter overflyttet til et maskinkodeprogram. Den fundne værdi skrives fra programmet til adressen, hvorfra den kan benyttes i BASIC.

Det efterfølgende program implementerer en udvidet PEEK-kommando med hjælp fra RSX. Med denne nye kommando er det muligt, at læse adresser i ROM, RAM og ekspansions-ROM (f.eks. diskette-ROM).

Kommandoen har følgende syntaks:

!RPEEK,@integervariabel, adresse,status

Således ville f.eks.:

!RPEEK,@w%,&C000,-7

loade værdien af den første adresse i diskette-ROM i variablen w%. Det er vigtigt at w% mindst er blevet brugt en gang tidligere i programmet, da ellers fejlmeddelelsen "IMPROPER ARGUMENT" vil fremkomme. For status gælder følgende:

1 : RAM
2 : ROM
0,-1,2 til -251 udvælger evt. ekspansions-ROM

Alle andre statusværdier medfører "IMPROPER ARGUMENT".

```
A000                10                          ; UDVIDELSE MED RSX
A000                20
; "!RPEEK,@INTVAR.,ADR.,STATUS
A000                30                          ; STATUS 1 : RAM
A000                40                          ; 2 : ROM
A000                50                          ; 0,-1,...,-251 : EXP. ROM
A000                60
A000                70    AOPMIS    EQU   &D055
; 464 : &CFEE / 664 : &D058
A000                80    AIMPAR    EQU   &C21D
; 464 : &C205 / 664 : &CB50
A000  010000        90    INIT      LD    BC,RSXTAB
A003  210000        100               LD    HL,SYSBYT
A006  CDD1BC        110               CALL  &BCD1
A009  3EC9          120               LD    A,&C9
A00B  3200A0        130               LD    (INIT),A
A00E  C9            140               RET
****  LINIE 90 :          RSXTAB=&A00F
A00F  0000          150   RSXTAB     DW    NAMTAB
A011  C30000        160               JP    START
****  LINIE 150:          NAMTAB=&A014
A014  52504545      170   NAMTAB     DM    "RPEE"
A018  CB            180               DB    &CB
A019  00            190               DB    0
****  LINIE 100:          SYSBYT=&A01A
A01A                200   SYSBYT     DS    4
A01E  DF            210   OPMIS      RST   &18
A01F  0000          220               DW    VEK1
A021  C9            230               RET
****  LINIE 220:          VEK1=&A022
A022  55D0          240   VEK1       DW    AOPMIS
A024  FD            250               DB    253
A025  DF            260   IMPARG     RST   &18
A026  0000          270               DW    VEK2
A028  C9            280               RET
****  LINIE 270:          VEK2=&A029
A029  1DC2          290   VEK2       DW    AIMPAR
A02B  FD            300               DB    253
****  LINIE 160:          START=&A02C
A02C  FE03          310   START      CP    3; TO PARAMETRE?
A02E  20EE          320               JR    NZ,OPMIS
; NEJ, SÅ MISSING OPERAND
```

```
A030  7A       330              LD    A,D
A031  FE00     340              CP    0
A033  28FE     350              JR    Z,NOEXRO
; STATUS > 0, SÅ INGEN EKSPANSIONS-ROM
A035  FEFF     360              CP    &FF ; VÆRDI < 255?
A037  20EC     370              JR    NZ,IMPARG
; JA, SÅ IMPROPER ARGUMENT
A039  7B       380              LD    A,E
A03A  ED44     390              NEG
A03C  FEFC     400              CP    252
A03E  30E5     410              JR    NC,IMPARG
A040  18FE     420              JR    SETSTA
****  LINIE 350:     NOEXRO=&A042
A042  7B       430  NOEXRO      LD    A,E
A043  FE03     440              CP    3
A045  30DE     450              JR    NC,IMPARG
A047  C6FE     460              ADD   A,254
A049  30FE     470              JR    NC,SETSTA
A04B  D604     480              SUB   4
****  LINIE 420:     SETSTA=&A04D
****  LINIE 470:     SETSTA=&A04D
A04D  320000   490  SETSTA      LD    (STATUS),A
A050  DD6E02   500              LD    L,(IX+2)
A053  DD6603   510              LD    H,(IX+3)
A056  DF       520              RST   &18
A057  0000     530              DW    VEKTOR
A059  DD6E04   540              LD    L,(IX+4)
A05C  DD6605   550              LD    H,(IX+5)
A05F  77       560              LD    (HL),A
A060  97       570              SUB   A
A061  23       580              INC   HL
A062  77       590              LD    (HL),A
A063  C9       600              RET
****  LINIE 530:     VEKTOR=&A064
A064  0000     610  VEKTOR     DW    ANFANG
****  LINIE 490:     STATUS=&A066
A066  FD       620  STATUS     DB    253
****  LINIE 610:     ANFANG=&A067
A067  7E       630  ANFANG     LD    A,(HL)
A068  C9       640              RET
```

```
PROGRAM: RPEEK
START:   &A000
SLUT:    &A068
LÆNGDE:  0069
FEJL:    0
```

VARIABELTABEL:

| | | | |
|---|---|---|---|
| AOPMIS D055 | AIMPAR C21D | INIT A000 | RSXTAB A00F |
| NAMTAB A014 | SYSBYT A01A | OPMIS A01E | VEK1 A022 |
| IMPARG A025 | VEK2 A029 | START A02C | NOEXRO A042 |
| SETSTA A04D | VEKTOR A064 | STATUS A066 | ANFANG A067 |

```
 10 FOR I=&A000 TO &A068
 20 READ A$:W=VAL("&H"+A$)
 30 S=S+W:POKE I,W:NEXT
 40 IF S <> 17592 THEN PRINT"FEJL I DATA":END
 50 ' 464: IF S<>17720 THEN.....
 60 ' 664: IF S<>17655 THEN.....
 70 PRINT"OK!":END
 80 DATA 01,0F,A0,21,1A,A0,CD,D1
 90 DATA BC,3E,C9,32,00,A0,C9,14
100 DATA A0,C3,2C,A0,52,50,45,45
110 DATA CB,00,20,A0,0F,A0,DF,22
120 DATA A0,C9,55,D0,FD,DF,29,A0
130 ' 464: .., ..,  EE,CF, .., .., .., ..
140 ' 664: .., ..,  58,D0,  .., .., .., ..
150 DATA C9,1D,C2,FD,FE,03,20,EE
160 ' 464: ..,  05,C2, .., .., .., .., ..
170 ' 664: ..,  50,CB, .., .., .., .., ..
180 DATA 7A,FE,00,28,0D,FE,FF,20
190 DATA EC,7B,ED,44,FE,FC,30,E5
200 DATA 18,0B,7B,FE,03,30,DE,C6
210 DATA FE,30,02,D6,04,32,66,A0
220 DATA DD,6E,02,DD,66,03,DF,64
230 DATA A0,DD,6E,04,DD,66,05,77
240 DATA 97,23,77,C9,67,A0,FD,7E
250 DATA C9
```

# 22. OPBYGNINGEN AF INDEKSEREDE VARIABLER

I dette kapitel vil vi beskæftige os med den interne lagring af indekserede variabler. Vi vil opbygge et maskinkodeprogram til opsummering af elementer.

Indekserede variabler lagres som normale variabler i en tabel efter BASIC-programmet. Tabellens startadresse står i adresse &AE87 (664/6128: &AE68). Da indekserede variabler kræver særdeles god lagerplads, skal disse DIMensioneres i forvejen med DIM-kommandoen. Indekserede variabler er, på samme måde som normale variabler, ordnet efter begyndelsesbogstav i kædede lister.

Tabellen er opbygget således:

Først står to bytes, der indeholder kæde-adressen. Derefter følger variablens navn, hvor sidste bogstav (ASCII-kode) adderes med 128 = &80, hvorved bit 7 sættes. Efter navnet følger type-betegnelsen. Så langt ligner de to slags variabler hinanden.

Derefter følger længden af hele indekset, der er lagret som et to-byte-tal. Derefter følger den maksimale størrelse for indekset.

Efter disse informationer følger rækken af værdier i indekset. Er det en integer-tabel, er der afsat to bytes for hvert element. Ved REAL-variabler har et element længden 5 bytes. Ved streng-variabler lagres også den 3 bytes lange string-descriptor.

Her følger en samlet oversigt over opbygningen:

| | |
|---|---|
| 2 bytes | Kæde-adressen |
| n bytes | variablens navn |
| 1 byte | Type-betegnelse |
| 2 bytes | Længden på hele indekset. |
| 1 byte | Antal elementer |
| 2 bytes | hvert indek's maksimale værdi |
| m bytes | værdien af hvert element. |

Længden 2 for INTeger
Længden 3 for strenge
Længden 5 for REAL

Det følgende program anvender denne struktur, for at opsummere antallet af elementer. Kommandoens syntaks ses i assemblerlistningen.

```
A000              10                        ; SUMMEN AF ET FELT
A000              20                        ; GIVER SUMMEN AF ALLE
                                            VÆRDIER
A000              30                        ; I EN TABEL
A000              40
A000              50
; SYNTAKS: CALL &A000,@variabelnavn(indices 0,..),ANTAL INDICES,
@variabel for resultat
A000              60
A000              70                  ORG   &A000
A000  DF          80                  RST   &18
A001  0000        90                  DW    VEKTOR
A003  C9          100                 RET
****  LINIE 90 :        VEKTOR=&A004
A004  0000        110   VEKTOR        DW    SUMAR
A006  FD          120                 DB    253 ; UROM ON
A007              130
A007              140
; ADR. FOR 6128 ; 464 , 664
A007              150   IMPARG        EQU   &C21D
; &FA9C , &CB50 IMPROPER ARGUMENT
A007              160   TYPMIS        EQU   &FF62
; &FF40 , &FF62 TYPE MISMATCH
A007              170   VARHND        EQU   &FF8C
; &FF66 , &FF87 VARIABEL (HL) TIL (DE)
A007              180   VARFAC        EQU   &FF6C
; &FF4B , &FF6C VARIABEL (HL) TIL FAC
A007              190   CPHLDE        EQU   &FFD8 ; &FFB8 , &FFD8 CP HL,DE
A007              200   READHD        EQU   &BD7C
; &BD58 , &BD79 REAL ARITM. (HL)+(DE)
A007              210   FAC           EQU   &B0A0 ; &B0C2 , &B0A0
A007              220   TYP           EQU   &B09F ; &B0C1 , &B09F
A007              230
****  LINIE 110 :       SUMAR=&A007
A007  D5          240   SUMAR         PUSH  DE
A008  210000      250                 LD    HL,FAC1 ; DE 5 BYTES
A00B  0605        260                 LD    B,5 ; I FAC1 SÆTTES
A00D  3600        270   NEXT          LD    (HL),0 ; TIL 0
A00F  23          280                 INC   HL
A010  10FB        290                 DJNZ  NEXT
A012  DD6E04      300                 LD    L,(IX+4)
A015  DD6605      310                 LD    H,(IX+5)
; ADRESSE FOR FØRSTE ELEMENT TIL HL
A018  DD7E02      320                 LD    A,(IX+2) ; ANTAL INDICES
A01B  47          330                 LD    B,A
A01C  E5          340                 PUSH  HL ; ADRESSE FOR FØRSTE
                                            ELEMENT
A01D  87          350                 ADD   A,A ; ANTAL INDICES * 2
A01E  C601        360                 ADD   A,1 ; FOR AT SUBTRAHERE TO
```

```
A020  1600    370            LD    D,0 ; BYTES FRA HL FOR HVERT
A022  5F      380            LD    E,A ; INDEKS
A023  B7      390            OR    A ; SLET CARRY
A024  ED52    400            SBC   HL,DE ; HL PEGER PÅ ANTALLET
A026  7E      410            LD    A,(HL); AF INDICES
A027  B8      420            CP    B ; SAMMENLIGNING
A028  C21DC2  430            JP    NZ,IMPARG
; HVIS ANTAL INDICES FORKERT : IMPROPER ARGUMENT
A02B  2B      440            DEC   HL ; FELTLÆNGDE
A02C  2B      450            DEC   HL ; SPRINGES OVER
A02D  2B      460            DEC   HL ; HL PEGER PÅ
A02E  7E      470            LD    A,(HL) ; PÅ TYPEKENDECIFFER
A02F  C601    480            ADD   A,1
A031  FE03    490            CP    3 ; HVIS STRENG, SÅ
A033  CA62FF  500            JP    Z,TYPMIS ; TYPE MISMATCH
A036  4F      510            LD    C,A
A037  329FB0  520            LD    (TYP),A
A03A  23      530            INC   HL
A03B  5E      540            LD    E,(HL) ; FELTETS LÆNGDE I
A03C  23      550            INC   HL ; BYTES TIL HL
A03D  56      560            LD    D,(HL)
A03E  19      570            ADD   HL,DE ; = SLUT PÅ FELTET
A03F  E3      580            EX    (SP),HL
; BYT START MED SLUT
A040  0600    590            LD    B,0
A042  F3      600            DI    ; FORDI DET ER HURTIGERE
A043  E5      610  WEITER    PUSH  HL ; ADRESSER NÆSTE
                                    ELEMENT
A044  C5      620            PUSH  BC ; TYPE I C
A045  CD6CFF  630            CALL  VARFAC ; VAR (HL) TIL FAC
A048  21A0B0  640            LD    HL,FAC
A04B  110000  650            LD    DE,FAC1
A04E  FE05    660            CP    5; REAL?
A050  28FE    670            JR    Z,REAL
A052  7E      680            LD    A,(HL)
A053  23      690            INC   HL ; HENT VÆRDIEN FOR DET
A054  66      700            LD    H,(HL) ; AKTUELLE ELEMENT
A055  6F      710            LD    L,A
A056  EB      720            EX    DE,HL
A057  7E      730            LD    A,(HL); HENT VÆRDIEN AF
                                    SUMMEN
A058  23      740            INC   HL
A059  66      750            LD    H,(HL)
A05A  6F      760            LD    L,A
A05B  19      770            ADD   HL,DE
A05C  220000  780            LD    (FAC1),HL ; FAC1=HL
A05F  18FE    790            JR    BREAK
****  LINIE 670 :    REAL=&A061
A061  EB      800  REAL      EX    DE,HL
```

```
A062  CD7CBD 810            CALL READHD ; REAL (HL)=(HL)+(DE)
****  LINIE 790 :    BREAK=&A065
A065  C1       820   BREAK  POP  BC ; TYPE I C
A066  E1       830          POP  HL ; SÆT HL PÅ NÆSTE
                             ADRESSE
A067  09       840          ADD  HL,BC ;
A068  D1       850          POP  DE; SLUTADRESSE
A069  D5       860          PUSH DE
A06A  CDD8FF   870          CALL CPHLDE ; SAMMENLIGN HL MED
A06D  38D4     880          JR   C,VIDERE
A06F  FB       890          EI   ; SLUT PÅ OPSUMMERING
A070  D1       900          POP  DE
A071  210000   910          LD   HL,FAC1
A074  D1       920          POP  DE ; RESULTAT KOPIERES
A075  CD8CFF   930          CALL VARHND ; TIL MÅLADRESSE
A078  C9       940          RET
****  LINIE 250 :    FAC1=&A079
****  LINIE 650 :    FAC1=&A079
****  LINIE 780 :    FAC1=&A079
****  LINIE 910 :    FAC1=&A079
A079           950   FAC1   DS   5

PROGRAM: SUMFELT
START:   &A000
SLUT:    &A07D
LÆNGDE:  007E
FEJL:    0
```

VARIABELTABEL:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| VEKTOR | A004 | IMPARG | C21D | TYPMIS | FF62 | VARHND | FF8C |
| VARFAC | FF6C | CPHLDE | FFD8 | READHD | BD7C | FAC | B0A0 |
| TYP | B09F | SUMAR | A007 | NEXT | A00D | WEITER | A043 |
| REAL | A061 | BREAK | A065 | FAC1 | A079 | | |

```
 10 REM BASIC LOADER FOR 6128
 20 REM PROGRAM SUMFELT
 30 FOR I=&A000 TO &A07D
 40 READ A$:W=VAL("&H"+A$)
 50 S=S+W:POKE I,W:NEXT
 60 IF S <> 15294 THEN PRINT "FEJL I DATA":END
 70 PRINT"OK!":END
 80 DATA DF,04,A0,C9,07,A0,FD,D5
 90 DATA 21,79,A0,06,05,36,00,23
100 DATA 10,FB,DD,6E,04,DD,66,05
```

```
110 DATA DD,7E,02,47,E5,87,C6,01
120 DATA 16,00,5F,B7,ED,52,7E,B8
130 DATA C2,1D,C2,2B,2B,2B,7E,C6
140 DATA 01,FE,03,CA,62,FF,4F,32
150 DATA 9F,B0,23,5E,23,56,19,E3
160 DATA 06,00,F3,E5,C5,CD,6D,FF
170 DATA 21,A0,B0,11,79,A0,FE,05
180 DATA 28,0F,7E,23,66,6F,EB,7E
190 DATA 23,66,6F,19,22,79,A0,18
200 DATA 04,EB,CD,7C,BD,C1,E1,09
210 DATA D1,D5,CD,D8,FF,38,D4,FB
220 DATA D1,21,79,A0,D1,CD,8C,FF
230 DATA C9,06,08,0D,20,CC
```

****

```
 10 REM SUMFELT FOR 464
 20 FOR I=&A000 TO &A07D
 30 READ A$:W=VAL("&H"+A$)
 40 S=S+W:POKE I,W:NEXT
 50 IF S <> 15372 THEN PRINT "FEJL I DATA":END
 60 PRINT"OK!":END
 70 DATA DF,04,A0,C9,07,A0,FD,D5
 80 DATA 21,79,A0,06,05,36,00,23
 90 DATA 10,FB,DD,6E,04,DD,66,05
100 DATA DD,7E,02,47,E5,87,C6,01
110 DATA 16,00,5F,B7,ED,52,7E,B8
120 DATA C2,9C,FA,2B,2B,2B,7E,C6
130 DATA 01,FE,03,CA,40,FF,4F,32
140 DATA C1,B0,23,5E,23,56,19,E3
150 DATA 06,00,F3,E5,C5,CD,4B,FF
160 DATA 21,C2,B0,11,79,A0,FE,05
170 DATA 28,0F,7E,23,66,6F,EB,7E
180 DATA 23,66,6F,19,22,79,A0,18
190 DATA 04,EB,CD,58,BD,C1,E1,09
200 DATA D1,D5,CD,B8,FF,38,D4,FB
210 DATA D1,21,79,A0,D1,CD,66,FF
220 DATA C9,06,08,0D,20,CC
```

****

```
 10 REM SUMFELT FOR 664
 20 FOR I=&A000 TO &A07D
 30 READ A$:W=VAL("&H"+A$)
 40 S=S+W:POKE I,W:NEXT
```

```
 50 IF S <> 15346 THEN PRINT "FEJL I DATA":END
 60 PRINT"OK!":END
 70 DATA DF,04,A0,C9,07,A0,FD,D5
 80 DATA 21,79,A0,06,05,36,00,23
 90 DATA 10,FB,DD,6E,04,DD,66,05
100 DATA DD,7E,02,47,E5,87,C6,01
110 DATA 16,00,5F,B7,ED,52,7E,B8
120 DATA C2,50,CB,2B,2B,2B,7E,C6
130 DATA 01,FE,03,CA,62,FF,4F,32
140 DATA 9F,B0,23,5E,23,56,19,E3
150 DATA 06,00,F3,E5,C5,CD,6C,FF
160 DATA 21,A0,B0,11,79,A0,FE,05
170 DATA 28,0F,7E,23,66,6F,EB,7E
180 DATA 23,66,6F,19,22,79,A0,18
190 DATA 04,EB,CD,79,BD,C1,E1,09
200 DATA D1,D5,CD,D8,FF,38,D4,FB
210 DATA D1,21,79,A0,D1,CD,87,FF
220 DATA C9,06,08,0D,20,CC
```

# 23. MASKINKODEPROGRAMMER I BEVÆGELSE: RELOKALISERING?

## 23.1. Hvorfor relokalisering?

Næsten alle kender problemet, hvad ententen man selv skriver maskinkodeprogrammer eller "låner" andres:

Programmerne fungerer udmærket hver for sig, men ønsker man at kombinere nogle rutiner, d.s.v. at flere programmer skal være tilgængelige samtidigt i computerens hukommelse, ja så er man på den. Det er især ved kombinationer af kommandoudvidelser, der opstår problemer p.g.a. ændringer af vektorer. Heldigvis, eksisterer dette problem ikke på CPC takket være den komfortable RSX (Resident System Expansion). Denne gør, at man praktisk talt kan kombinere alle de kommandoudvidelser, man har lyst til. RSX-systemet behøver ikke at ændre nogen af operativsystemets vektorer; operativsystemet er forberedt for eksterne kommandoer.

Det andet problem, der opstår ved kombination af maskinkodeprogrammer, skyldes ikke operativsystemet, men selve maskinsproget: Et maskinkodeprogram er i princippet bundet til den del afhukommelsen, hvori det er skrevet. Den kan flyttes til et vilkårligt sted i hukommelsen, men kan kun bringes til udførelse på det sted, hvor den oprindelig er skrevet. Dette skyldes, at mange af maskinkodekommandoerne benytter sig af absolutte adresser. De kommandoer, der i et program, er rettet mod adresser i det samme område, som programmet benytter, kan naturligvis kun udføres eet sted.

Da de fleste kommandoudvidelser befinder sig så langt oppe som muligt i hukommelsen, for at tage så lidt som muligt af den kostbare BASIC-RAM, opstår der et besværligt problem, når kommandoer skal kombineres i dette område. Oftest vil programmerne kollidere og overlappe hinanden, hvilket jo som bekendt er fatalt for systemet.

## 23.2. Hvordan relokalisering fungerer

Trylleordet, der kan klare vore problemer angående kombination af maskinkodeprogrammer, hedder relokalisering. Bag dette velklingende fremmedord gemmer der sig en teknik, der gør et maskinkodeprogram uafhængig af selvadresserende og absolutte adresser. For at kunne opnå det, er der forskellige muligheder:

Den første af disse muligheder beror på anvendelsen af specielle "mellemfiler", der indeholder en særlig loader-kode, der ved siden af det egentlige program, også indeholder vigtige oplysninger for relokaliseringen. Loader-koden kan ikke udføres som et maskinkodeprogram. Den skal flyttes til adresseområdet via et specielt loader-program. Når det er flyttet hertil, vil det blive konverteret til et køreklart program.

Det til denne opgave bestemte loader-program, er ikke en almindelig loader, men en såkaldt LINKER. Foruden at loade og starte et program, kan en linker også sammenføje flere relokative program-moduler. Det er f.eks. denne teknik firmaet MicroSoft har anvendt til MACRO80-ASSEMBLER. Den tilhørende linker kan foruden at starte det færdige program, gemme det på diskette eller i hukommelsen, også implementere rutiner fra et bibliotek. Der kan hentes data fra flere programmer samtidigt, idet der anvendes globale labels.

Blandt de nævnte muligheder, er denne metode uden tvivl den bedste. Der gives her en fleksibilitet, der kun begrænses af hukommelsplads, regnehastighed og iderigdom. Større maskinkodeprogrammer kan med denne software-pakke gøres særdeles overskuelige, strukturerede og brugervenlige.

Man kan også dele programmer og data op i afsnit, der kan placeres forskellige steder i hukommelsen, hvilket er særdeles praktisk, hvis man selv brænder sine EPROM's.

For alle disse fordele må man betale en temmelig høj pris: Der kræves en speciel assembler, der understøtter den særlige loader-kode. Denne assembler er ikke kun væsentlig større end en normal assembler, men p.g.a. af sin kompleksitet og udvidede anvendelse også en del vanskeligere at betjene/udnytte i fuldt omfang. Linkeren er heller ikke, hvad man kalder et mindre program.

Den anden mulighed tager udgangspunkt i et program skrevet med en normal assembler. Programmet bindes af assembleren til en (i princippet) bestemt basisadresse (for test af programmet, anbefales det, at man lægger det et sted, hvor det kan udføres). Nu er det, at knoklearbejdet begynder: Der skal bruges en, specielt til dette program beregnet, programloader, der beder om den ønskede basisadresse, eller på anden måde erfarer denne. Den skal også kunne ændre de adresser, programmet selv bestemmer, og tilpasse det til de nye adresser.

Den egentlige loader ser ganske normal ud og kan tilmed uden videre skrive i BASIC; dog skal tabellen med de adresser, der skal tilpasses, skrives manuelt for hvert program. Denne tabel er allerede ved mellemstore programmer, særdeles omfangsrig. En automatisk fremstilling af denne tabel, f.eks. via et hjælpeprogram, der kan hente sine informationer fra notater i assembler-source-koden, er tænkelig, men desværre uden for denne bogs rammer.

Der findes endnu en metode, der dog er af mere eksperimentiel art. Det drejer sig om en såkaldt "automatisk relokalisator", der disassemblerer det aktuelle program og foretager en ændring af adressen afhængigt af om en kommando, der indeholder en absolut adresse, er en del af selve programmet. (16-bits load-kommandoer eller hopkommandoer med udvidet, eller udvidet og umiddelbar, adressering.). Metoden er bekvem at anvende, men desværre ikke så pålidelig: Så snart der i et program findes en adresse afhængig af programmets adressetabel, fungerer den ikke længere. Desuden er en værdi, der står efter en 16-bits load-kommando, ikke nødvendigvis en adresse, selvom den måske svarer til en adresse i programmet. Tæller-startværdier og andre konstanter ændres uden hensyntagen til sådanne relokalisatorer.

Af disse grunde, kan det nok siges at være et tilfælde, hvis et program af bare nævneværdigt omfang viser sig at fungere, som det skal.

## 23.3. Det kan gøres på en nemmere måde!

Den første af de i det forrige afsnit nævnte metoder til relokalisering er temmelig vidt udbredt - hvilket man ikke kan sige om den sidste metode! Begge metoder har alligevel nogle afgørende ulemper: Den første fordrer anskaffelse af et bestemt, ikke billigt program og en indarbejdning af et nyt system til assemblerprogrammering. Den anden metode kræver en del "håndarbejde".

Den metode, vi vil vise her, har ikke meget til fælles med de tre nævnte. I det følgende vil vi vise en metode, der med få ændringer, kan bruges til næsten alle relokaliseringsopgaver. Det anvendte maskinprogram er kun 42 bytes langt. Det princip, der anvendes er næsten genialt! Problemet med uflytbare maskinkodeprogrammer ville næsten ikke eksistere, dersom der fandtes ekvivalenter til de relative hop-kommandoer (JR) og JP og CALL-kommandoer (med 16-bits hop-længde, for at kunne nå hele hukommelsen rundt) samt til load-kommandoerne med udvidet, eller udvidet og umiddelbar adressering. Og det er netop, hvad vi har tænkt os at finde!

Hvis disse kommandoer er en så stor mangel, så lad os implementere dem. "Opfindelse" af nye kommandoer er ikke noget, man bare gør; og derfor vil vi gå lidt mere i detaljer m.h.t. fremgangsmåden.

Ved konstruktionen af de nye kommandoer, vil vil gøre brug af en metode, som fremkommer som et tilbud fra Z80-processorens konstruktører, nemlig PREFIX-metoden. Den går ud på, at man tager en allerede defineret Opcode, og sætter et tal foran (et såkaldt prefix). Derved får koden en ny betydning. Hvis man f.eks. tager en kommando, der benytter HL-registeret og sætter prefix'et &DD foran, så får vi en kommando til IX-registeret, eller via &FD til IY-register. Via den nye prefix (der endnu ikke er fundet) skal der ændres en kommando med en 16-bits-operand, således at operanden ikke bliver absolut, men relativ for den aktuelle PC-værdi (program-counter).

Vi kan naturligvis ikke ændre Z80-processorens funktion for at indføre vore egne kommandoer. I stedet skal vi bruge et lille hjælpeprogram (42 bytes langt), der kaldes i det øjeblik vores relokaliserings-prefix dukker op. Som prefix benytter vi CPC-operativsystemets instruktion RST &30, der netop er reserveret til brugerformål (svarer til CALL &0030). Det ville også kunne lade sig gøre at bruge simple CALL-kommandoer, men det indebærer visse ulemper, som f.eks. udførelseshastigheden og udnyttelse af hukommelsen. RST-instruktionens fordel er, at den i modsætning til CALL-kommandoen, praktisk talt fungerer uden angivelse af hop-adresse og kun optager en byte.

Instruktionen RST &30 sættes sammen med kommandoen, der skal fortolkes relativt. Inden den egentlige kommando udføres, møder processoren instruktionen RST &30 og udfører det tilhørende underprogram. Denne rutine "plukker" nu den relative adresse fra den tilhørende kommando (hvor kommandoen befinder sig, fremgår af den i STACK stående retur-adresse for RST-kommandoen) og beregner den effektive adresse ud fra basisadressen og retur-adressen. Den beregnede adresse flyttes til kommandoen. RST-kommandoen overskrives med en NOP-instruktion, og der returneres til det egentlige program, hvor den endelige kommando udføres.

Metoden gør, at programmet så snart det er blevet udført, igen befinder sig (resident) i sit eget adresseområde. Dette har dog ikke nogen praktisk betydning, idet det kun yderst sjældent forekommer, at et program skal flyttes under udførelse.

## 23.4. Relokaliseringsprogrammet

Så er det slut med den grå teori for denne gang. Her følger listningen af vort hjælpeprogram. Den tilhørende funktionsbeskrivelse tjener til at gøre det teoretiske grundlag for metoden, lidt klarere.

Her er listningen:

```
0000 E3       RELRST EX   (SP),HL   ; GEM HL, RETURADRESSE
                                      TIL HL
0001 D5              PUSH DE         ; GEM DE
0002 C5              PUSH BC         ; GEM BC
0003 F5              PUSH AF         ; GEM AF
0004 E5              PUSH HL         ; GEM RETURADRESSE
0005 7E              LD   A,(HL)     ; KOMMANDOS 1. BYTE
0006 FE ED           CP   &ED        ; TEST FOR PREFIX
0008 28 08           JR   Z,SKPPFX
000A FE DD           CP   &DD
000C 28 04           JR   Z,SKPPFX
000E FE FD           CP   &FD
0010 20 01           JR   NZ,NOPFX
0012 23       SKPPFX INC  HL         ; SPRING OVER PREFIX
0013 23       NOPFX  INC  HL         ; HL PEGER PÅ ADRESSEFELT
0014 5E              LD   E,(HL)     ; OFFSET LO-BYTE
0015 23              INC  HL
0016 56              LD   D,(HL)     ; OG HENT HI-BYTE
0017 2B              DEC  HL         ; HL PEGER PÅ START AF
                                      ADRESSE FELTET
0018 EB              EX   DE,HL      ; POINTER PÅ ADRESSEFELT
                                      TIL DE, OFFSET TIL HL
0019 C1              POP  BC         ; RETURADRESSE (=BASIS-
                                      ADRESSE) TIL STACK
001A C5              PUSH BC         ; OG IGEN TIL STACK
001B 09              ADD  HL,BC      ; ADDER OFFSET TIL BASIS
001C EB              EX   DE,HL      ; ADRESSE-FELT-POINTER
                                      TIL HL, EFFEKTIV ADRESSE
                                      TIL DE
001D 73              LD   (HL),E     ; EFF. ADR. LO-BYTE
001E 23              INC  HL
001F 72              LD   (HL),D     ; SKRIV HI-BYTE TIL
                                      ADRESSEFELT
0020 E1              POP  HL         ; RETURADRESSE TIL HL
0021 2B              DEC  HL         ; HL PEGER PÅ RST
```

```
0022  36  00        LD    (HL),&00  ; ERSTAT RST MED NOP
0024  23           INC   HL        ; HL INDEHOLDER IGEN
                                     RETURADRESSE
0025  F1           POP   AF        ; HENT AF
0026  C1           POP   BC        ; HENT BC
0027  D1           POP   DE        ; HENT DE
0028  E3           EX    (SP),HL   ; HENT HL,RETURADRESSE
                                     TIL STACK
0029  C9           RET             ; RETUR-HOP
```

Nogle af læserne vil måske have lagt mærke til, at denne assemblerlistnings basisadresse ligger i adresse &0000. Hvilket ved første blik betyder, at programmet ikke vil kunne udføres. RAM-området fra &0000 til &003F er absolut tabu (fy fy) for brugerprogrammer, fordi det indeholder de vigtigste og flittigst benyttede vektorer for operativsystemet (en undtagelse er vektoren for RST &30). Denne kendsgerning er dog ikke nogen hæmsko, idet det drejer sig om relative adresser. Det vil sige, at de ikke repræsenterer de endelige adresser, men derimod afstanden til en endnu ikke defineret basisadresse.

Angivelsen af relative adresser er uden værdi for relokative assemblere; der er jo ikke den fjerneste ide i at angive absolutte adresser, når den egentlige basisadresse er ukendt.

Bryder læseren sig ikke om adresseangivelsen, kan han/hun blot forestille sig et A i stedet for det første 0! Hvordan den endelige basisadresse fremkommer, vil man erfare i det følgende afsnit.

Trods programmets ringe omfang, opfylder det i enhver henseende de stillede krav.

Iøvrigt, skal det nævnes, at da det drejer sig om en kommandoudvidelse til Z80-processoren, skal man passe på, at der ikke sker ændringer af registrene. Hvis programmet skal være helt sikkert, må det ikke øve ekstern indflydelse på noget register. Derfor gemmes indholdet af samtlige registre i starten af programmet (PUSH) og hentes tilbage fra STACK igen, inden retur-hop (POP).

*Læg iøvrigt mærke til, at CPC er nødt til at slå den nederste ROM fra, for at kunne nå RST-kommandoens vektor.*

## 23.5. Loader-programmet

Inden vi går i gang med en indgående funktionsbeskrivelse, skal vi lige have afklaret et par spørgsmål:

Hvordan flyttes programmet hen til det sted, hvor det skal bringes til udførelse?

Hvordan sikres det, at relokaliseringsprogrammet aktiveres ved kald af RST &30?

Lad os starte med at besvare det sidste spørgsmål. Bortset fra at kommandoen RST &30 kun er 1 byte lang, så svarer den nøje til kommandoen CALL &0030. Fra adressen &0030 i RAM står der 8 bytes til rådighed for brugeren, fra &0030 til &0037, der kan anvendes til kaldet af RST-rutinen. Da der næppe findes nogen rutine, der har plads nok i 8 bytes, skal der her placeres en vektor, der peger på den ønskede rutine. Denne består af en hop-instruktion (JP). Det er altså kun den tilsvarende Opcode (&C3) til &0030 der skal flyttes til måladressen for vort program (&0031-&0032).

Hvordan findes BASIS-adressen og hvordan finder programmet derhen?

Det er ved skrivningen af vort hjælpeprogram i hukommelsen, vi får glæde af, at den er fuldstændig relokativ. Vi kan nu uden problemer skrive en loader til programmet, der automatisk flytter det hen, hvor det er mest gunstigt, at placere det: I øverste RAM. For loadning af programmet, benytter vi en variant af den ukomplicerede BASIC-loader:

```
 10 REM BASIC-LOADER FOR RELOKALISERINGS-PROGRAM
    VERSION 1.0
 20 REM (HR) 11/85
 30 MEMORY HIMEM-42
 40 FOR MPTR=HIMEM+1 TO HIMEM+42
 50 READ BYTE
 60 POKE MPTR;BYTE
 70 CS=CS+BYTE
 80 NEXT
 90 IF CS <> &H165C THEN PRINT CHR$(7);
    "FEJL I TESTSUM ";HEX$(CS):STOP
100 REM RST-VEKTOR SÆTTES
110 POKE &H0030,&HC3:POKE &H0031,(HIMEM+1) AND 255:
    POKE &H0032,(HIMEM+1)/256
120 PRINT"RELOKALISERINGSPROGRAM IMPLEMENTERET
    VED ";HEX$(HIMEM,4)
130 END
1000 DATA &HE3,&HD5,&HC5,&HF5,&HE5,&H7E,&HFE,&HED
1005 DATA &H28,&H08,&HFE,&HDD,&H28,&H04,&HFE,&HFD
1010 DATA &H20,&H01,&H23,&H23,&H5E,&H23,&H56,&H2B
1015 DATA &HEB,&HC1,&HC5,&H09,&HEB,&H73,&H23,&H72
1020 DATA &HE1,&H2B,&H36,&H00,&H23,&HF1,&HC1,&HD1
1025 DATA &HE3,&HC9
```

Programbeskrivelse:

30: Der reserveres plads i øverste RAM ved sænkning af HIMEM.

40-80: Maskinkoden læses fra DATA-linierne og lagres i det reserverede område. I linie 70 opsummeres testsummen.

90: Her kontrolleres testsummen, og i givet fald skrives en fejlmeddelelse ud (hvis testsummen ikke er korrekt) og programmet afbrydes.

110: Flytning af RST &30-vektoren til startadressen for hjælpeprogrammet.

120: Load-processens udfald meddeles med angivelse af basisadresse.

130: Her afsluttes programmet. I stedet for END-kommandoen, kan man indsætte NEW-kommandoen, således at det efter brugen unødvendige BASIC-program, ikke længere optager plads i hukommelsen.

## 23.6. Relokaliseringsprogrammets praktiske anvendelse

Hvordan skal et assemblerprogram se ud, for at kunne arbejde sammen med relokaliserings-hjælperutinen? De ændringer, der skal til i forhold til et normalt assemblerprogram er minimale. I selve programmet skal følgende ændringer foretages:

* Hver kommando, der skal benytte en i operand-tabellen liggende adresse, skal foranstilles RST &30.

* I de samme kommandoer, skal den absolutte adresse erstattes af en PC-relativ adresse af 16 bits længde. En PC-relativ adresse angiver ikke afstanden fra basisadressen, men derimod afstanden til selve kommandoens udførelseslocus (som JR-kommandoen).

Hvordan med PC-relative adresser? Denne adresseringsmetode er uden særlige dikkedarer. Anvendelsen af PC-relative kommandoer er særlig fordelagtig i vort tilfælde, da programmet henter alle sine data direkte eller indirekte via STACK. Distancen på 16 bits er i alle henseender tilstrækkelig adresseplads.

Indlæsningen af en PC-relativ adresse i et assemblerprogram er absolut ukompliceret. Adressen skal blot tilføjes et "$" . Vi forestiller os følgende underprograms-kald, der står inden i et program (kaldet angår en adresse, der benyttes af selve programmet):

```
CALL NUMOUT
```

For at konvertere denne kommando, skal der i stedet for skrives:

```
RST  &30
CALL NUMOUT-$
```

Assembleren gør, at man kan accessere en kommando's adresse ved at tilføje et "$". Kommandoen:

```
JP   $
```

svarer til en uendelig løkke.

BEMÆRK: Tilgang til PC er muligt i de fleste assemblere. Dollartegnet ($) er det hyppigst benyttede symbol hertil. Afvigende betegnelser, som f.eks. "*" kan forekomme.

## 23.7. Et relokativt eksempel

Nu vil vi bruge vort program til relokalisering i et praktisk eksempel. Det programmet skal udføre, er ikke overvældende. Efter et kald med CALL tæller det baglæns fra 100 til 0, hvorefter det vender tilbage til BASIC.

Selvom programmet ikke har nogen egentlig funktion, bortset fra dets værdi som demonstrationsprogram, består det dog alligevel af en del interessante rutiner. Programmet er nemlig, bortset fra rutinen TXT OUTPUT, skrevet fuldstændig uden brug af operativsystemet. Det indeholder enddog sin egen decimale udskrivningsrutine: NUMOUT. Denne rutine kalder rutinen DIV168, der udfører en binær division. Vi vil kigge nærmere på rutinerne, da de er gode eksempler på, hvordan komplicerede aritmetiske rutiner, kan laves.

Her følger listningen for programeksemplet:

```
; DEMO FOR RELRST

0000  21 00 65  RRDEMO  LD    HL,101        ; STARTVÆRDI+1
0003  2B        LOOP00  DEC   HL            ; TÆL NED
0004  E5                PUSH  HL            ; GEM TÆLLER-
                                               INDHOLD
0005  F7                RST   &30           ; RELOKALISERINGS-
                                               PREFIX
0006  CD 0D 00           CALL  NUMOUT-$      ; UDLÆS TÆLLER-
                                               STAND
0009  E1                POP   HL            ; HENT TÆLLERSTAND
000A  7C                LD    A,H           ; HL
000B  B5                OR    L             ; = 0?
000C  20 F5             JR    NZ,LOOP00     ; NEJ, TÆL VIDERE...
000E  3E 0D             LD    A,0DH         ; UDSKRIV CR
0010  C3 5A BB          JP    TXTOUT        ; YXY OUTPUT

; UDSKRIV 16-BIT-TAL I HL PÅ SKÆRM
; ÆNDREDE REGISTRE AF, HL, DE, BC, IX

0013  E5        NUMOUT  PUSH  HL            ; GEM TAL FOR
                                               UDLÆSNING
0014  F7                RST   &30
0015  21 4B 00           LD    HL,OUTBUF-$   ; SLET OUTPUT-BUFFER
0018  F7                RST   &30
```

```
0019  11 48 00          LD    DE,OUTBUF+1-$
001C  01 00 05          LD    BC,5
001F  36 20             LD    (HL),' '
0021  ED B0             LDIR
0023  E1                POP   HL            ; HENT TAL FOR
                                               UDLÆSNING
0024  F7                RST   &30
0025  DD21 3F 00         LD    IX,OUTBUF+4-$ ; IX PEGER PÅ
                                               OUTPUT-BUFFER
0029  0E 0A             LD    C,10          ; DIVISOR (KONSTANT)
002B  F7        LOOP01   RST   &30
002C  CD 1D 00           CALL  DIV168-$      ; DEL HL MED 10
002F  C6 30              ADD   A,'0'         ; REST --> ASCII-TAL
0031  DD77 00            LD    (IX),A        ; SKRIV CIFFER I
                                               BUFFER
0034  DD2B               DEC   IX
0036  EB                 EX    DE,HL         ; KVOTIENT ER NYT TAL
0037  7C                 LD    A,H           ; TAL
0038  B5                 OR    L             ; =0?
0039  20 F0              JR    NZ,LOOP01     ; NEJ, KONVERTER
                                               VIDERE...
003B  F7                 RST   &30
003C  21 24 00           LD    HL,OUTBUF-$   ; UDLÆS BUFFER
003F  06 05              LD    B,5           ; TÆLLER FOR TEGN
0041  7E        LOOP02   LD    A,(HL)        ; HENT TEGN
0042  CD 5A BB           CALL  TXTOUT        ; UDLÆS TEGN
0045  23                 INC   HL            ; NÆSTE TEGN
0046  10 F9              DJNZ  LOOP02        ; VIDERE, HVIS EJ
                                               FÆRDIG
0048  C9                 RET                 ; SLUT PÅ UDLÆSNING

; DIVIDER 16 BIT MED 8 BIT
; INDLÆS: DIVIDEND I HL, DIVISOR I C
; UDLÆS : KVOTIENT I DE, REST I A
; ÆNDRET REGISTER: AF, HL, DE, BC

0049  11 00 00 DIV168    LD    DE,0          ; KLARGØR REGISTER
                                               FOR KVOTIENT
004C  06 10              LD    B,16          ; BIT-TÆLLER
004E  AF                 XOR   A             ; KLARGØR REGISTRE
                                               FOR JOB OG REST
004F  CB 23     LOOP03   SLA   E             ; KVOTIENT
0051  CB 12              RL    D             ; FORSKYD MOD
                                               VENSTRE
0053  CB 25              SLA   L             ; DIVIDEND
0055  CB 14              RL    H             ; FORSKYD MOD
                                               VENSTRE
0057  17                RLA                 ; MEST BETYDENDE BIT
                                               I AKK
```

```
0058  B9              CP    C          ; SUBTRAKTION MULIG?
0059  38 02           JR    C,SKIP00   ; NEJ, SPRING OVER
005B  13              INC   DE         ; FORHØJ KVOTIENT
005C  91              SUB   C          ; SUBTRAHER DIVISOR
005D  10 F0  SKIP00   DJNZ  LOOP03     ; IKKE SLUT, VIDERE
005F  C9              RET
0060         OUTBUF   DEFS  5          ; 5 BYTES OUTPUT-
                                          BUFFER
```

I demonstrationsprogrammets listning findes de relative adresser. I programmet har alle kommandoerne, der har en absolut adresse indenfor programmet selv, fået foranstillet RST &30. Adresserne er konverteret til PC-relative adresser ved tilføjelsen -$. Eksempler på dette finder man i kommandoerne (adresserne &0014, &0018, &0024 og &003C), der gør brug af OUTBUF (output-buffer). Det viser sig ved, at også anvendelse af Offset (f.eks. OUTBUF+4) kan lade sig gøre. Load-kommandoerne, der ikke indeholder adresser, der er indeholdt i selve programmet, som f.eks. loadningen af konstanterne ved &0000, ændres ikke. Kommandoen ld ix,outbuf+1-$ i adresse &0024 demonstrerer, hvorledes RSX-kommandoudvidelsen med kommandoer, som indeholder et index-register-prefix. RST-kommandoen foranstilles prefix. Hjælpeprogrammet detekterer dette og tager hensyn til det ved tilgang til den aktuelle kommandos adressefelt (se adresserne &0005 til &0013 i listningen af relokaliseringsprogrammet). CALL-kommandoerne for underprogrammet NUMOUT og DIV168 er ligeledes forsynet med RST &30. Kald af rutinen TXT OUTPUT er derimod ikke blevet behandlet, da rutinen jo er knyttet til operativsystemet.

De relative hop-kommandoer JR og DJNZ er selvfølgelig heller ikke blevet behandlet. De er heller ikke knyttet til absolutte adresser.

Devisen "HELLERE EN GANG FOR MEGET END EN GANG FOR LIDT" gælder for relokaliseringsprogrammet. Unødvendige placerede RST &30-kommandoer vil øve indflydelse på de efterfølgende kommandoer, og vil som regel resultere i at computeren hænger sig op.

Slut med teorien og videre til en BASIC-loader, hvormed man kan afprøve demonstrationsprogrammet med forskellige udførelsesadresser:

```
 10 INPUT "BASIADRESSE ";BASIS
 20 OLDHIMEM=HIMEM
 30 MEMORY BASIS-1
 40 FOR MP=BASIS TO BASIS+100
 50 READ BYTE
 60 POKE MP,BYTE
 70 CS=CS+BYTE
 80 NEXT
 90 IF CS <> &2751 THEN PRINT "FEJL I TESTSUM!":STOP
100 CALL BASIS
110 PRINT"SLUT!!"
```

```
 120 MEMORY OLDHIMEM
 130 END
1000 DATA &21,&65,&00,&2B,&E5,&F7,&CD,&0D
1005 DATA &00,&E1,&7C,&B5,&20,&F5,&3E,&0D
1010 DATA &C3,&5A,&BB,&E5,&F7,&21,&4B,&00
1015 DATA &F7,&11,&48,&00,&01,&05,&00,&36
1020 DATA &20,&ED,&B0,&E1,&F7,&DD,&21,&3F
1025 DATA &00,&0E,&0A,&F7,&CD,&1D,&00,&C6
1030 DATA &30,&DD,&77,&00,&DD,&2B,&EB,&7C
1035 DATA &B5,&20,&F0,&F7,&21,&24,&00,&06
1040 DATA &05,&7E,&CD,&5A,&BB,&23,&10,&F9
1045 DATA &C9,&11,&00,&00,&06,&10,&AF,&CB
1050 DATA &23,&CB,&12,&CB,&25,&CB,&14,&17
1055 DATA &B9,&38,&02,&13,&91,&10,&F0,&C9
1060 DATA &00,&00,&00,&00,&00
```

Denne BASIC-loader beder om en basis-adresse, hvorefter den reserverer plads og flytter (kopierer) demonstrationsprogrammet derhen. I den nye HIMEM sættes med MEMORY-kommandoen, bliver den gamle HIMEM-værdi gemt for at begyndelstilstanden kan genoprettes efter at maskinkodeprogrammet er afsluttet.

For at programmet kan bringes til udførelse, skal relokaliseringsprogrammet installeres. Med BASIC-loaderen kan man placere programmet, hvor man ønsker det, hvilket i princippet vil sige fra &1000 til &9FFF. Pas på at loadningen af demoprogrammet ikke overskriver relokaliseringsprogrammet! Den højeste basisadresse for demo-programmet er altså relokaliseringsprogrammets basisadresse. Overholder man grænserne, kan man placere sit demo-program hvor man ønsker det. Fjerner man RST &30-kommandoerne, kan man implementere programmet, som et ganske normalt maskinkodeprogram. M.h.t. hastighed, vil man erfare, at der faktisk ikke eksisterer nogen forskel i udførelsestid.

På nuværende tidspunkt burde det ikke være vanskeligt for læseren selv, at skrive programmer for relokaliseringsprogrammet.

## 23.7.1. Demo-programmets underrutiner

Som nævnt i det forrige afsnit, vil vi kigge lidt nærmere på demonstrationsprogrammet underrutiner:

NUMOUT og DIV168

Vi kan selvfølgelig ikke fremvise en komplet afhandling om aritmetikrutiner i dette kapitel. Vi nøjes med et kort rids af rutinernes funktion, for at hjælpe læseren til at skrive egne rutiner. Vi håber, at denne gennemgang og kommentarerne i listningerne kan give et godt indblik i rutinernes funktion.

Rutinerne kan (særligt divisionsrutinen) overtages som de står anført i listningen, uden ændringer, til brug i egne rutiner. Kommentarerne i listningen giver oplysninger om ud- og indlæsningsparametre og ændringer i de berørte registre. Begge rutiner kaldes med de simple CALL-kommandoer.

OUTPUT-rutinen NUMOUT

Denne rutine udlæser de i register HL indlæste 16-bits binære tal i decimal form. Udlæsningen foregår højreordnet i et felt bestående af 5 tegn fra den aktuelle cursorposition. Ikke benyttede udlæsningspositioner fyldes ud med mellemrumstegn (ASCII 32). Tallet behandles uden fortegn, d.v.s. at heltalsværdier i intervallet 0 til 65353 kan udlæses.

Rutinen benytter for udlæsning den 5 tegn (bytes) lange output-buffer OUTBUF. Det skyldes ikke blot den højrestillede udlæsning, men også fordi denne rutines tal opbygges bagfra. De enkelte cifre skrives med start i laveste position og udlæses samlet.

OUTPUT-rutinen starter med at fylde BLANKS i output-bufferen (adresserne &0013 til &0023). Dette sker med (fuldt overlæg) forkert brug af kommandoen LDIR. Kommandosekvensen :

```
LD    HL,START
LD    DE,START+1
LD    BC,LAENGE-1
LD    (HL),BYTE
LDIR
```

kan en blok med adressen START og længden LAENGE og værdien BYTE fyldes ud, idet blok kopierer sig selv. Det er en relativ hurtig (hvis ikke den hurtigste) metode til at udfylde et bestemt område med en bestemt byte-sekvens (også en enkelt byte).

Derefter initialiseres register IX som pointer for slutningen på bufferen. Register C loades med den for konvertering nødvendige konstant 10.

Derefter begynder den egentlige konvertering fra label LOOP01. Til dette formål gentagent divideret med 10 (DIV168) og den derved opståede divisionsrest lagres i bufferen (LD(IX),A) efter konvertering til ASCII-ciffer (ADD A,'0'). Herefter flyttes buffer-pointeren et tegn frem (DEC IX). Den i register DE overflyttede heltals-kvotient (via divisionsrutinen), lagres som nyt tal i HL (EX DE,HL) inden der startes på et nyt gennemløb af løkken.

Denne proces gentages indtil kvotienten efter et løkkegennemløb er 0. Den her anvendte null-test for 16-bits-register med kommandoerne LD og OR er en simpel og hurtig testmetode for registrene HL, DE og BC, er især i forbindelse med 16-bit-INC og DEC-kommandoer ideel, da disse kommandoer ikke påvirker nogen flag.

Er kvotienten 0, så er der ikke flere ciffre, der skal findes. Selve konverteringen er afsluttet og tallet befinder sig højreordnet i bufferen med foranstillede BLANKS. Det samlede bufferindhold udskrives i en løkke (LOOP02) via systemrutinen TXT OUTPUT på skærmen.

For at gøre udskrivningsrutinens arbejdsgang lidt mere forståelig, følger her på baggrund af konverteringen af tallet 523, indholdet af de resulterende registre og buffere ved de enkelte gennemløb af løkken LOOP01 fra adresse &0036. Indholdet af registrene er angivet i decimal form og indeholdet af bufferen svarer til de tilhørende ASCII-tegn. Indholdet af register HL bliver, idet den er ændret af divisionsrutinen, ikke virkelig angivet i adresse &0036, men derimod inden kald af divisionsrutinen angivet i adresse &002B.

| GENNEMLØB | HL | DE | A | BUFFER |
|---|---|---|---|---|
| 1 | 523 | 52 | 3 | 3 |
| 2 | 52 | 5 | 2 | 23 |
| 3 | 5 | 0 | 5 | 523 |

Efter det tredie gennemløb, er kvotienten i DE blevet 0 og konverteringsløkken kan bringes til afslutning. I slutningen af rutinen vil der blive returneret til det kaldende program via RET.

Divisionsrutinen DIV 168

Denne rutine dividerer et binært 16-bit-tal med et 8-bit-tal og overfører en 16-bits-kvotient og returnerer en 8-bits divisionsrest. Alle tal er hele tal uden fortegn.

For at forstå denne rutine, må man have et nært kendskab til binær aritmetik. Da denne bog ikke danner grundlag for et sådant kendskab, må vi desværre undlade at komme nærmere ind på det emne.

Her skal kun nævnes at princippet er det samme, som ved den skriftlige division. En binær division er en af de mest komplicerede elementære rutiner i maskinsprog. På baggrund af nogle eksempler på papir, kan man gennemspille rutinen og eksperimentere.

## 23.8. Grænserne for denne metode til relokalisering

I dette afsnit skal vi sætte grænserne for den viste metode til relokalisering.

Med vores lille relokaliseringsprogram, kan vi uden problemer behandle tilgangen til et objekt, hvis adresse ligger i en kommandos operandfelt. Det bliver mere kompliceret, dersom vi vil gribe til et objekt via en adresse i f.eks. en tabel bestående af hopadresser. Men selv dette problem kan løses.

Til det formål må maskinprogrammet finde ud af, hvor det selv befinder sig i hukommelsen. Det følgende underprogram skal kaldes:

```
GETPC  POP   HL
       JP    (HL)
```

Efter kald med CALL GETPC henter underprogrammet via POP-kommandoen, sin returadresse fra STACK og springer til denne via JP(HL). Det har samme effekt som en simpel RET-kommando, men giver i tilgift returadressen, d.v.s. adressen direkte efter CALL-kommandoen i HL. Dermed har vi næsten, hvad vi skal bruge. Med sekvensen:

```
       RST   &30
       CALL  GETPC-$
BASE   RST   &30
       LD    (BASADR-$),HL
```

kan den egentlige absolutte adresse for label BASE findes. Tilføjelse til en tabel (f.eks. en hop-tabel), kan kun angives relativt til BASE. I stedet for:

```
       DW    adresse
```

skal man skrive

```
       DW    adresse-BASE
```

i tabellen. Inden access til objektet, addere BASE adressen, der opbevares i 16-bit-hukommelsen BASADR, til den relative adresse. Hvis vi antager at den relative adresse befinder sig i HL, kan det f.eks. ske med følgende sekvens:

```
RST   &30
LD    DE,(BASADR-$)
ADD   HL,DE
```

De dertil nødvendige underprogrammer og adresseplads, kræver så lidt plads, at de ikke skulle kunne forstyrre andre ting. I vore eksempler har vi, så vidt muligt, gjort brug af relokaliserings-prefixet (RST &30).

Sammen med de her angivne tillægs-oplysninger kan man praktisk talt klare alle relokative opgaver, der gør brug af adresser indenfor samme program. Tilgang til et adresseområde benyttet af et andet program er nok en stor opgave, idet vi ikke råder over global tilgang.

## 23.9. Loadning af et relokativt program

BASIC-loaderen rækker til loadning af mindre programmer, som f.eks. demo-programmet. BASIC-loaderen kan uden større anstrengelser fremstilles ved hjælp af et

program. Loaderen kan dermed laves, så den erfarer basisadressen på baggrund af HIMEM, som det er tilfældet med loaderen for relokaliseringsprogrammet.

For længere programmer er dette ikke tilrådeligt, da længden af BASIC-loaderen vokser temmelig hurtigt (den er ca. 3 gange så omfattende som det egentlige program). Her er det bedre med en loader, der kan indlæse en på basisadressen &0000 beliggende binærfil relokativt. Her er det også på sin plads at konstatere basisadressen via HIMEM.

# ORDLISTE (ikke komplet)

## A

### Acces

Den tid, der går fra data kaldes og indtil de er tilgængelige. Kaldes også tilgangstid.

### Adresse

Talkode, der betegner et bestemt sted i hukommelsen.

### Algoritme

En fuldstændig beskrevet fremgangsmåde til løsning af et givet program.

### Arbejdslager

Det område af hukommelsen, der bruges til lagring og bearbejdning af data. Kaldes også RAM-området.

### ASCII

En amerikansk standard for udveksling af bogstaver og symboler. ASCII er en kodetabel, hvori alle bogstaver, tal og specialtegn har fået en talkode. Denne talkode er international. Oprindelig udviklet til brug for telex-kommunikation.

## B

### BASIC

Beginners All-purpose Symbolic Instruction Code. Sikkert det mest udbredte og mest anvendte programmeringssprog på hjemmmecomputere og minicomputere (PC-ere).

### BCD-format

Tallene 0-9 noteret i total-system. Hvert tal udgøres af 4 cifre (bits).

### Bloksøgning

En gruppe af data søges.

### Bloktransfer

En gruppe af data overføres.

### Brugerinterface

Kobling mellem bruger og datamat.

### Buffer

Mellemlager i en datamat.

## Bug (lus)

Programfejl.

# C

## Cartridge

Et modul, der indeholder en ROM-kreds hvori et program er lagret. De fleste hjemmecomputere er forsynet med en såkaldt port, hvori modulet (cartridge) forbindes med computeren. Fordelen ved at bruge en cartridge som lagermedium i stedet for bånd eller diskette, er den ultrakorte tid, det tager at udføre et program fra ROM-kredsen i modulet til computerens arbejdslager. Det er især spil og alternative programmeringssprog, der forhandles i cartridge-form.

## Compiler

Er betegnelsen for en "oversætter". Et højniveausprog som BASIC skal før det kan "køre" omdannes til maskinsprog som computeren kan forstå. I normal BASIC sker dette løbende efterhånden som programmet afvikles. Denne form for oversættelse kaldes fortolkning, men i stedet for at anvende denne relativt langsomme metode, kan man oversætte sit program een gang for alle. Det er det, man skal bruge en compiler til.

## Cursor

Den lille markør på skærmen, der viser hvor "man" befinder sig. Markøren kan have flere former, og kan enten være blinkende eller fast. Cursoren svarer til spidsen af en blyant.

# D

## Database

Indholdet i alle filer.

## Datatransmission

Data sendes ad elektronisk vej fra et sted til et andet.

## Diskette

En rund magnetiserbar polyesterskive indkapslet i et plasthylster. Ydre lagringsmedie for en computer. Kapaciteten kan variere fra under 100 Kbytes til over 1 Mbytes.

# F

## Fil

En ordnet samling af sammenhørende data. I daglig tale refereres til den fysiske tilstedeværelse af et EDB-program/dataoplysninger på en diskette eller et bånd.

## Formattering
Den proces, der inddeler en diskette i et antal spor og sektorer for skrivning og læsning af data.

# H

## Hexadecimal
Talsystem, der har 16 som grundtal.

# I

## Implementere
At udføre det arbejde der kræves for at gøre teori til praksis.

## Indexerede rækker
Rækker der har fået index.

## Initialisere
At tildele en variabel en værdi på forhånd.

## Inversere
Omvende.

# K

## Kompatible
To datamater siges at være kompatible, hvis de kan afvikle det samme program uden ændringer, kommunikere med hinanden eller bruge de samme data.

## Konfiguration
Betegnelse for indholdet af de enkelte enheder i et samlet computer-anlæg. Enhederne omfatter både hukommelse, interfaces og tilslutningen af printere, diskettestationer, monitorer.

## Krympe
Formindske.

## Konverteres
Omvende eller omforme.

# L

## Label
Identifikation, mærke eller etiket.

## Load

Kommando der bevirker at et bestemt program indlæses fra baggrundslager.

## Løkke

Programdel, hvis instruktioner skal gentages, indtil en bestemt betingelse er opfyldt.

# M

## Markør

Se under CURSOR.

## Matrix

Rektangulær opstilling af elementer i række inden for matematikken.

# O

## Operand

Data, som kan behandles ved hjælp af en operator.

## Overflow (stakoverløb)

Datamatens kapacitet (lager) er opbrugt.

# P

## Parametre

En variabel, hvis værdi er fast i en given anvendelse.

## Pixel

Et enkelt element i et billede/tegn.

## Port

Dataindgang til centralenheden (CPU). Bruges ofte som betegnelse for alle ud- og indgange på en computer.

## Printer

En slags elektrisk skrivemaskine uden tastatur. Computeren kan aktivere printerens skrivehoved, således at de enkelte bogstaver og tegn kan udskrives på papir.

## Processor

Del af central enheden i en datamat.

## Pseude-akkumulator

Kunstig-akkumulator.

# R

## Rutiner

Program eller dele af et program, der anvendes almindeligt eller hyppigt.

# S

## Scrollning

Rulning (skærmbilledet).

## Sekventielt

Data umiddelbart efter hinanden i en fil.

## Simulering

Efterligning af virkelige forhold ved hjælp af en model.

## Sound-chippen

Lyd-chippen.

## Streng-variabler

Navnet på en streng (følge af bogstaver, tal og/eller tegn).

## String (alfanumeriske data)

Følge af tal, tegn eller bogstaver.

## Syntax

Regler for dannelse af tilladelige konstruktioner i et sprog.

# V

## Valid tegn

Lovligt tegn.

## Vektor

Numerisk variabel med en dimension.

## 3-D-grafik

Tre-dimentionalt grafik.

**Ovenstående ordliste er, som angivet i overskriften, ikke komplet, men indholdet er et skønsomt antal af de fremmedord, som optræder i bogen. Skulle der, efter din opfattelse være glemt nogle der burde have været med, hører vi gerne herom SKRIFTLIGT.**

# AMSTRAD-bladet...

## -dit brugerblad

**Med alt det spændende, nye til en af de allerstærkeste computere på markedet.**

Amstrad-bladet er det eneste officielle brugerblad i Danmark.
Dette betyder at DU som bruger bliver holdt orienteret om alle de mange spændende projekter der sker omkring Amstrad computeren, både i Danmark og i udlandet, via direkte telexforbindelse til producenten i England samt vort samarbejde med den danske importør, Dinamicro.
Foruden nyheder og reportager bringer Amstrad-bladet også masser af tests (så du undgår at købe »katten i sækken«), oplysende artikler om programmering (så din investering måske kunne give en lille ekstraindtægt), 16 sider med programmer, som er lige til at taste ind, annoncer, tips og tricks, læser til læser – kontakt, oplysninger om brugerklubber og meget, meget mere.
Bladet er trykt i 4-farve offset og udkommer hver anden måned. Hvert nummer indeholder minimum 40 sider.
Amstrads computere har over hele verden vist sig at være virkelige 'top – chart – runner's når det gælder de internationale salgslister. Men selv en så stærk computer som Amstrad'en har brug for opbakning. Dinamicro og Twilights forhandlere er indstillede på at yde dig den bedst mulige hjælp vedr. dit køb og igangsætningen, resten af vejen kan Amstrad-bladet være en god og værdifuld kilde til oplysninger og interessante opgaver.

**EN YDERLIGERE FORDEL FOR DIG:**

**Få vort nye blad «Input», fyldt med programlistninger til din Amstrad - Helt Gratis...**

Vi vil gerne have lov at præsentere vort nye programblad for alle nye ejere af Amstrad computere. Derfor får du gratis det første nummer tilsendt, når du tegner abonnement, og så håber vi naturligvis, at du bliver SÅ begejstret at du henter **INPUT** hos din bladhandler, hver gang det udkommer



FORLAGET MICROTECH
Gødvad Bakke 4 8600 Silkeborg 06 82 24 55