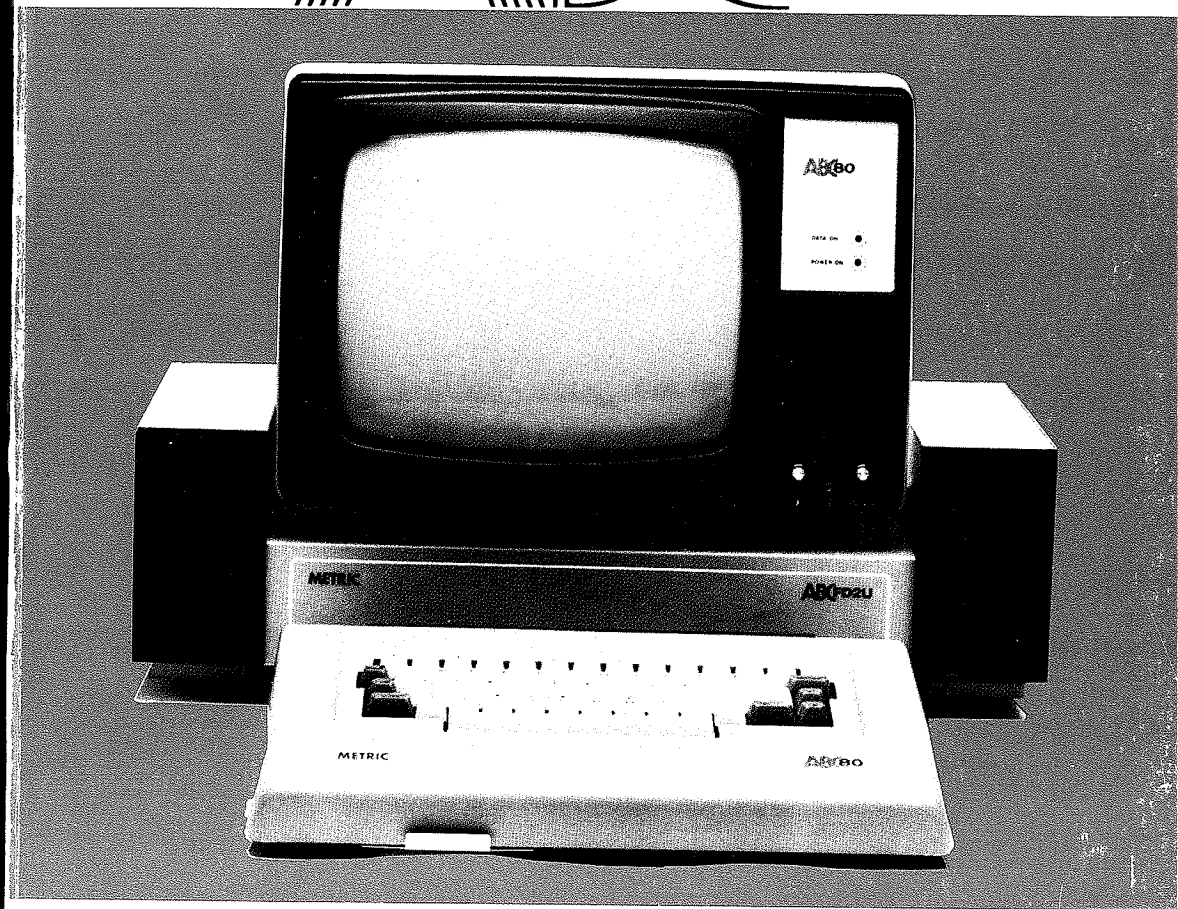
Printed in Sweden
Studentlitteratur
Lund 1980
ISBN 91-44-17451-9

Innehåll

Avancerad programmering på ABC 80



Anders Isaksson Örjan Kärrsgård

Avancerad programmering på ABC 80

Anders Isaksson Örjan Kärrsgård

SCANDIA METRIC AB Studentlitteratur

Printed in Sweden
Studentlitteratur
Lund 1980
ISBN 91-44-17451-9

Innehåll

KAPITEL 1

Inledning

ABC80

Den här boken riktar sig till dig som redan kan en del om datorer och vill veta hur man gör professionella program på ABC80. Du kanske vill gå vidare efter att ha lärt dig det grundläggande om ABC80, eller du känner till hur andra datorsystem fungerar och vill nu lära känna ABC80 och dess möjligheter.

Boken kan sägas bestå av en BASIC-del och en assembler-del. Fram till och med kapitel 7 programmerar vi i BASIC. Vi börjar med att gå igenom god metodik för programmering. Därefter behandlas dataelement, filer och människa-dator interaktion. När vi kommit så långt är det dags att använda våra kunskaper, vilket vi gör i ett större exempel.

Kapitel 8 till 10 behandlar BASIC-interpretatorn, assemblerprogrammering och anslutningstekniker för yttre enheter.

Kapitel 11 slutligen behandlar datakommunikation, hur datorer kan samarbeta över tele- och data-nät. Ett viktigt område.

Den utrustning vi använder är en ABC80 grundenhet (tangentbord och skärm) samt en diskettenhet (FD2). För att underlätta programmeringen bör någon form av skrivare även ingå.

Något om beteckningar:

Vi använder i den här boken "byte" som beteckning på en bitgrupp om 8 bitar (oktad). För att referera en bit inom en byte används beteckningen b0 till b7. Den minst signifikanta biten är b0. Detta

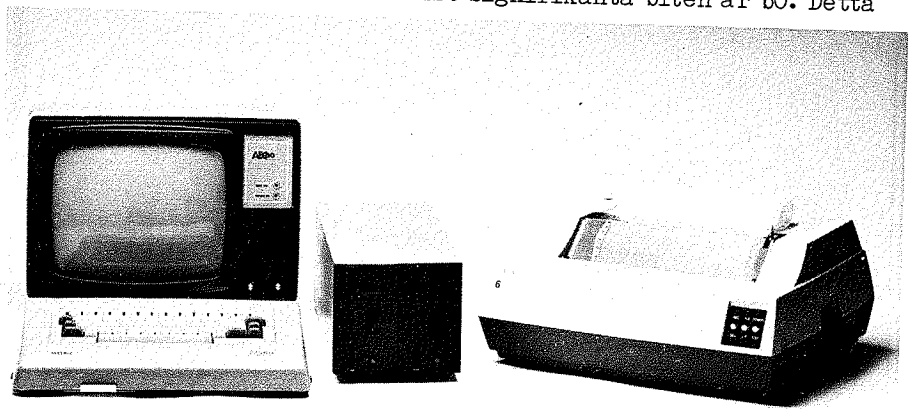


Fig 1.1

överensstämmer med de manualer från Zilog som bör användas som brevidläsning i kapitel 8, 9 och 10.
(Svensk standard är dock beteckningen b1 till b8 på bitarna i en oktad).

I appendix-A finns en tabell över decimaltal, binärtal och hexadecimaltal. Ett binärtal anger vi med talet åtföljt av "B" och ett hexadecimaltal åtföljt av "H".

Den skrivare som denna bok är tryckt på saknar tecknet "ö", istället används "$".

KAPITEL 2

Programkonstruktion

ABC80

2.1 Systemutvecklingsetapper

Vi kan använda vår ABC80 på många olika sätt. Ibland är det en enstaka beräkningsuppgift vi önskar lösa och vi skriver då raskt ett program för detta. Vi har då använt vår ABC80 närmast som en programmerbar kalkylator.

I andra fall kan det handla om program som används under en längre tid, och av helt andra personer än den som har skrivit programmet. Det kan t ex röra sig om ett mätsystem som används vid tillverkningen av en produkt eller om order-, lager- och faktureringsprogram för ett företag.

Vi ska först närmare titta på den senare typen av tillämpningar och vilken standard, vilka modeller och metoder som kan hjälpa oss utforma ett bra system.

Vad menas då med ett bra system ?

Det ska vara vara enkelt att använda för den som arbetar med systemet. En bruksanvisning måste finnas till hands etc.

Systemet ska vara lätt att underhålla. Det ska vara ändringsvänligt. Utformningen och dokumentationen ska göra det möjligt att införa rättelser och ändringar - även för någon annan än den som ursprungligen gjorde programmet.

Huvudmotiven för att införa ADB är oftast, i någon mening, ekonomiska. Systemet är inte bra förrän intäkten av systemet överstiger kostnaden av att utveckla och införa detsamma.

Låt oss nu uppehålla oss lite kring begreppet standard.

När det gäller t ex en mekanisk konstruktion finns en mängd standard. Det gäller både ingående komponenter och hur maskinen ska beskrivas i dokumentationen. Ritningsunderlag måste finnas i form av detaljritningar, sammanställningsritningar, ritningsförteckning, detaljförteckning etc. Även hur en ritning ska se ut finns det standard för.

Vid t ex en ändring kan verkstadschefen ganska lätt tala om både hur lång tid det tar och vad det kommer att kosta.

Eftersom datorer inte funnits så länge har man inte kommit lika långt vad beträffar standard för programkonstruktioner. Sätten

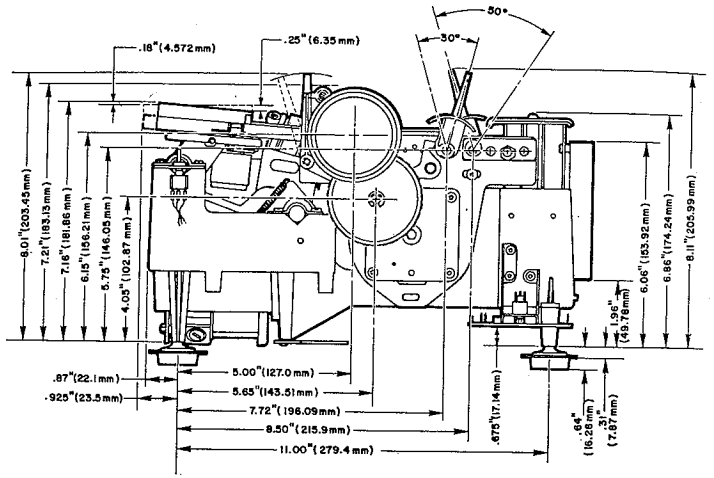


Figure 2-3 HYTYPE II DIMENSIONS, END VIEW

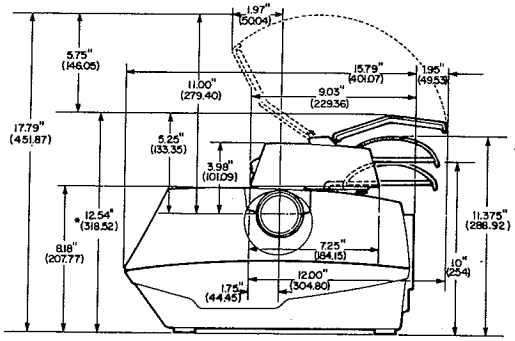
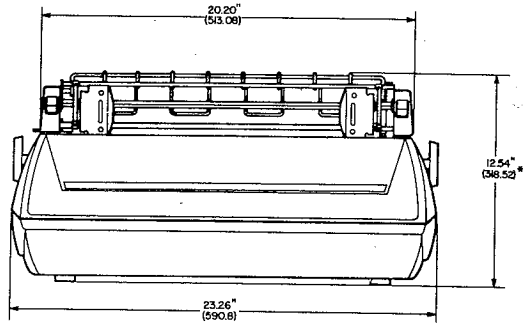


Figure 2-4 HYTYPE II DIMENSIONS, END VIEW WITH COVERS AND FORMS TRACTOR



NOTES: ALL DIMENSIONS ARE NOMINAL,
DIMENSIONS SHOWN IN PARENTHESES ARE IN MILLIMETERS
* = ADJUSTABLE

Figure 2-5 HYTYPE II DIMENSIONS, FRONT VIEW WITH COVERS AND FORMS TRACTOR

Fig 2.1 Exempel på mekanisk ritning (diablo-skrivare).

att utveckla och konstruera program har länge varit, snart sagt lika många som antalet programmerare.

Vad finns då att gå efter idag ?

Sveriges Standardiseringskommision (SIS) har givit ut en handbok: Riktlinjer för administrativ systemutveckling (RAS). Boken börjar med några synpunkter på projektorganisation. Därefter presenteras en modell för systemutveckling. Modellen är uppbyggd av 8 etapper:

1. Målstudie
2. Informationsstudie
3. Behandlingsstudie
4. Systemstudie
5. Detaljstudie
6. Detaljutformning
7. Systeminförande
8. Efterstudie

En grövre etappindelning kan vara analys, konstruktion och realisering.

För varje etapp i RAS finns en checklista på vad som bör göras i varje etapp och var etappernas arbete ska dokumenteras.

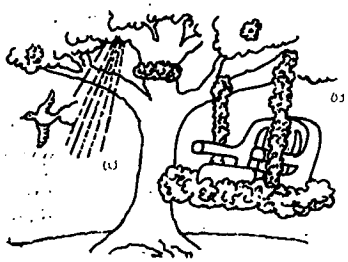
I den här boken, Avancerad programmering på ABC80, kommer vi mest att intressera oss för etapperna 5 och 6.

Skulle du som läser dessa rader komma att delta i ett ADB-projekt, utan att ha gjort det tidigare, kanske följande råd kan vara på sin plats.

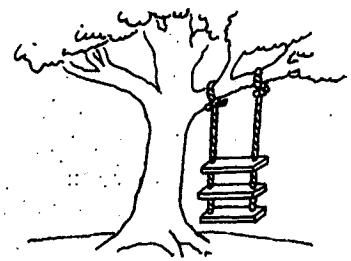
1. Tro inte att du begriper någonting om systemering bara för att du kan BASIC ! (Hoppas du blev förargad).

2. Ha inte allt för stor ambition i början. Gör en etappindelning och börja med att lägga över någon enstaka rutin på data.

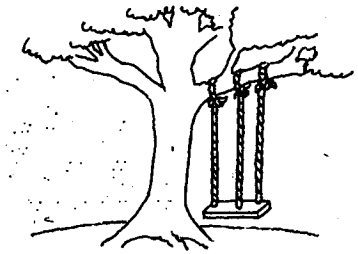
3. Genomför den första etappen i projektet så tidigt som möjligt för att få en omedelbar återkoppling från användarna. De kommer att ha helt andra synpunkter efter att ha provat praktiskt, jämfört med att ha läst/hört specifikationer.



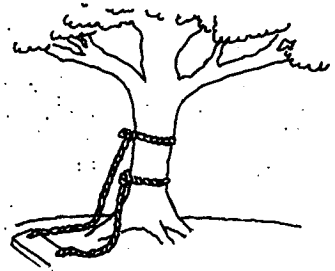
Vad försäljarna lovade kunden.



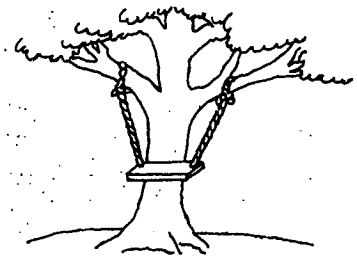
Hur man preciserade kundens önskemål.



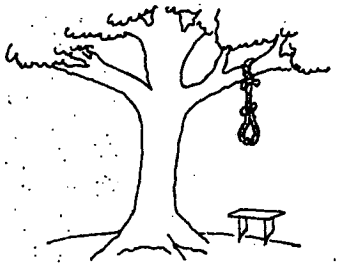
Hur man ansåg att systemet skulle fungera.



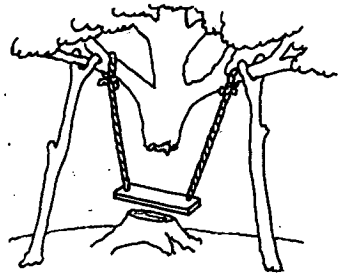
Hur systemmännen realiserade systemets funktion.



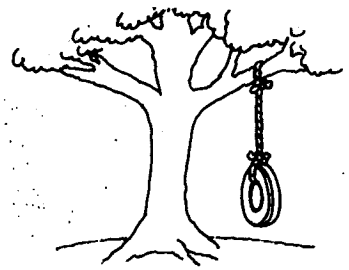
Vad programmeraren producerade.



Vad systemansvariga såg i systemet.



Vad man installerade hos kunden.



Vad kunden egentligen ville ha.

4. Tillägna dig en god metodik. Programmering är ett hantverk som kräver erfarenhet. Mycket "onödigt lidande" har drabbat användare av datorsystem därför att systemfolk och programmerare inte behärskat sitt hantverk.

Man diskuterar ofta vad som är bäst: Stora centrala datorsystem - eller små dedicerade system typ ABC80. Titta på checklistan nedan och fundera över detta!

1. Vad är bakgrunden och motivet till att utveckla och införa datoranvändning/nytt system.

2. Vem är ansvarig för utvecklingen.

3. Vem ska leda utvecklings- och projektarbetet.
   - är det en dataexpert,
   - någon från linjen,
   - någon utomstående.

4. Vad kommer det att kosta
   - köps färdigt system utifrån,
   - standardsystem,
   - egen dator,
   - servicebyrå etc.

5. Finns det alternativa lösningar som inte kräver datoranvändning.

6. Vilka enheter/avdelningar berörs.

7. Förändras beslutsprocessen
   - decentralisering, centralisering
   - möjlighet till inflytande.

8. Vilken typ av system är det fråga om
   - varaktighet.

9. Kommer systemet att vara flexibelt, utbytbart
   - ingår det något eller några standardsystem.

10. Vem berörs av systemen.

11. Sker förändringar i arbetsuppgifter
   - arbetstakt,
   - försvinner krav på yrkeskunskaper,
   - utarmning,

   - inflytande över den egna arbetssituationen,
   - positiva möjligheter (arbetutvidgning, rotation, överblick etc).

12. Påverkas sysselsättningen.
   - minskar, ökar,
   - heltid, deltid,
   - omflyttning, omplacering,
   - skiftgång,
   - arbetstidens längd och förläggning,
   - övertid.

13. Kommer det att innebära utbildnig, omskolning
   - intern, extern,
   - för hur många.

14. Förändras löneformerna
   - ackord, fast lön,
   - individuella prestationer,
   - meritvärdering,
   - begripligt löneunderlag.

15. Den personliga integriteten
   - olika register,
   - registrens innehåll,
   - individuppgifter,
   - krävs tillstånd från Datainspektionen.

16. Konsekvenser för arbetsmiljön
   - monotoni, stress,
   - risk för olycksfall,
   - ergonomiska aspekter,
   - social isolering.

Låt oss efter denna utvidgning återgå till programkonstruktion. Vi ska i nästa avsnitt gå igenom en metod som bidrar till att få program på ABC80 väl strukturerade och dokumenterade.

2.2 Strukturerad programmering

2.2.1 ALLMÄNT OM JSP

Den metod för programkonstruktion som vi nu ska gå igenom har fått sitt namn efter den person som hittat på den, Michael Jackson. Metoden heter "Jackson Structured programming" eller kort JSP. Det skulle föra för långt att här gå igenom alla tankegångar, ideer och möjligheter i JSP. Vi nöjer oss med att gå igenom det grundläggande och med ett större exempel i kapitel 6 ge förståelse för JSP.

Innan du själv börjar använda JSP, vilket vi rekommenderar, bör du läsa boken "JSP - en praktisk metod för programkonstruktion" (Ref-8). Läs den!

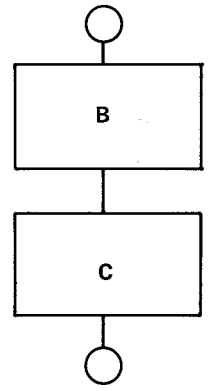
Först måste vi göra klart för oss skillnaden mellan programkonstruktion och kodning. Det är inte särskilt lämpligt att vi så fort vi förelagts ett problem sätter oss och knappar in BASIC-satser. Vi bör istället KONSTRUERA programmet först och sedan låta koden "trilla ut" mer eller mindre automatiskt.

JSP bygger på ett speciellt sätt att rita flödesplaner eller snarare strukturdiagram. En viktig poäng i JSP är att man börjar med att beskriva de DATA som programmet ska arbeta med i en datastruktur. Denna leder fram till en programstruktur. Följande steg genomförs:

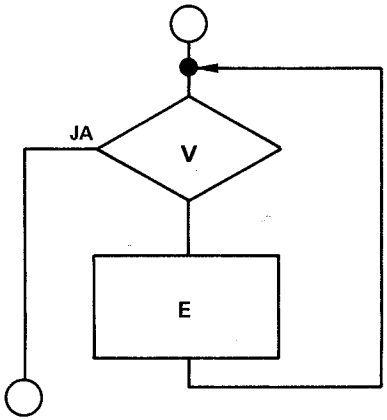
1. Rita datastrukturer
2. Förena datastrukturer till programstruktur.
3. Lista operationer och placera dessa i programstrukturen.
4. Koda

2.2.2 TRE PROGRAMKOMPONENTER

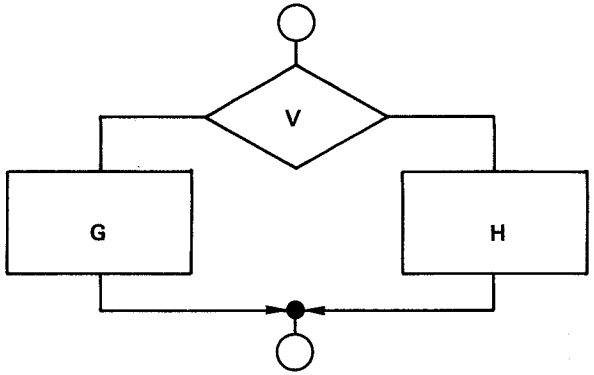
För att skriva ett program behövs bara tre programkomponenter ! Dessa är SEKVENS, ITERATION och SELEKTION. Med vanlig flödesplanbeskrivning ser dessa ut så här:



1. SEKVENS: Först B, sedan C.



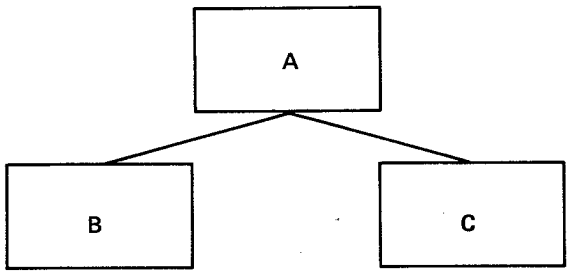
2. ITERATION: E till dess villkor V är uppfyllt. (Noll eller flera gånger).



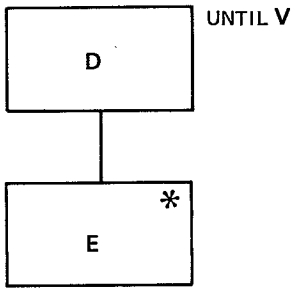
3. SELEKTION: Antingen G eller H.

Fig 2.3

Med JSP-notation ser dessa istället ut så här:



1. SEKVENS: Först B, sedan C.



2. ITERATION: E till dess villkor V är uppfyllt.
(Noll eller flera gånger).

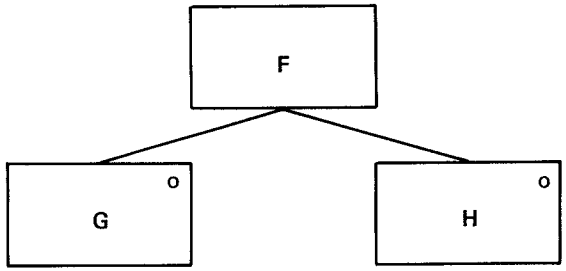


Fig 2.4

3. SELEKTION: Antingen G eller H.

Med dessa tre komponenter kan vi beskriva såväl data som program. En enkel beskrivning av den här boken (data) kan t ex se ut så här:

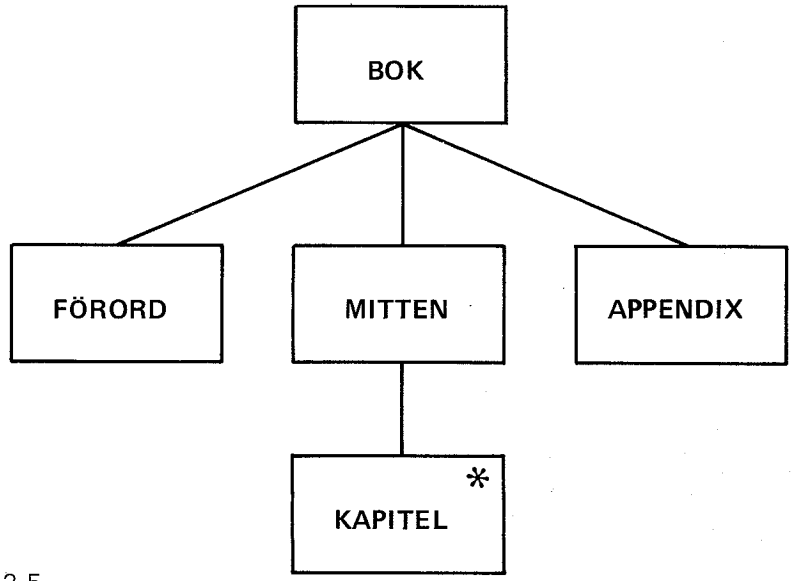


Fig 2.5

BOK är en sekvens av FÖRORD, MITTEN och APPENDIX. Mitten är en iteration av KAPITEL.

2.2.3 KODNING I BASIC

JSP är i sig språkoberoende. Vi bör dock ha några regler så att man lätt kan hitta programstrukturen i programlistan. I språk med symboliska lägesnamn, t ex COBOL eller ABC80-assembler, kan vi ge läget namn efter JSP-rutan (t ex A-SEQ:). I BASIC är vi hänvisade till att använda kommentarer.

Sekvens:

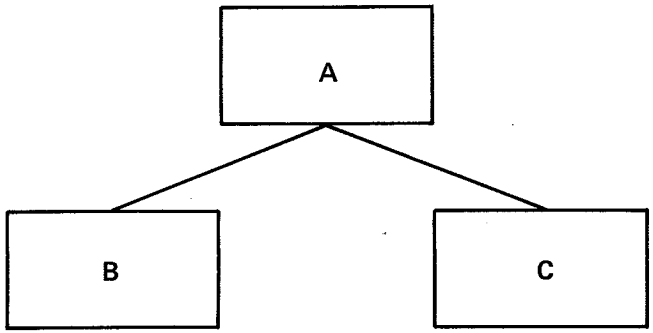


Fig 2.6

```
100 REM *A-seq*
110 GOSUB 430 : REM *B*
120 GOSUB 450 : REM *C*
130 REM *A-end*
```

B och C kan antingen vara subrutiner eller kodas "rakt" på stället.

Iteration:

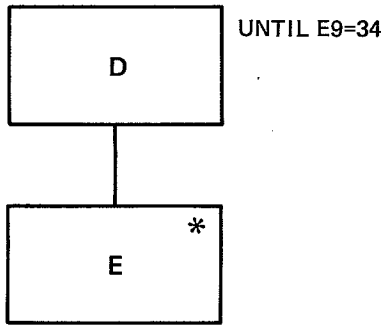


Fig 2.7

```
200 REM *D-ite*(E9=34)
210 IF E9=34 THEN 240
220 GOSUB 500 : REM *E*
230 GOTO 200
240 REM *D-end*
```


Observera att satsen som testar på termineringsvillkoret ligger först i iterationen. Iterationen (mera vanvördigt: loopen) kan därmed även löpas igenom noll gånger, d.v.s ingen gång alls om villkoret redan är uppfyllt. Raden före slutraden (230) kommer i en iteration alltid att bestå av en GOTO till iterationens början.

För den enkla typ av iterationer som brukar kallas räkne-loopar är naturligtvis konstruktioner med FOR och NEXT tillåtna.

Selektion:

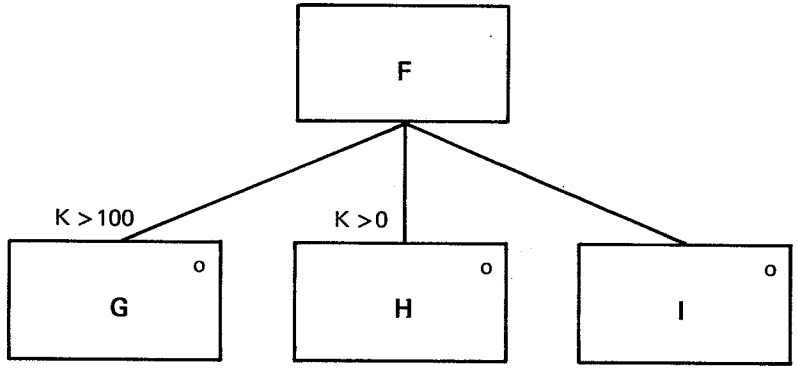


Fig 2.8

```
300 REM *F-sel*(K>100)
310 IF NOT (K>100) THEN 340
320 GOSUB 600 : REM *G*
330 GOTO 400
340 REM *F-or1*(K>0)
350 IF NOT (K>0) THEN 380
360 GOSUB 650 : REM *H*
370 GOTO 400
380 REM *F-or2*
390 GOSUB 680 : REM *I*
400 REM *F-end*
```

Observera att selektionsvillkoren gås igenom från vänster till höger. G kommer att utföras om K>100, H om 100>=K>0 och I i övriga fall (K negativt). För att allt ska fungera måste selektionsvillkoret vändas, rad 310 och 350, så att hopp sker till nästa ruta om villkoret INTE stämmer. Efter koden för rutorna G och H finns ett GOTO till selektionens slut.

Den sista rutan måste alltid vara villkorslös. Något av alternativen måste alltid utföras. En JSP-ruta kan dock även vara tom, d.v.s. ingenting utförs. En sådan ruta markeras med ett streck.

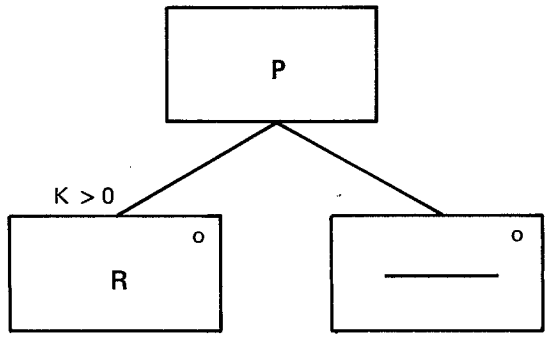


Fig 2.9

En av de stora fördelarna med JSP är att man alltid på förhand vet vart alla GOTO-satser leder. Man kan med okontrollerade GOTO-satser annars lätt trassla till det för sig (och andra!), så att programmet snarast får karaktären av "ormbo" eller "spagetti".

I appendix F finns ett komplett program konstruerat och dokumenterat med JSP.

Vi ska gå igenom detta senare men tjuvtitta gärna redan nu!

2.3 Programdokumentation

2.3.1 ALLMÄNT

Då man följer JSP-metoden kommer en stor del av dokumentationen fram redan i konstruktionsfasen.

Förutom programdokumentationen krävs en bruksanvisning som talar om för användaren hur programmet används. Hur bruksanvisningen (ibland kallad handhavandebeskrivning) ska se ut är det svårt

att ge fasta regler för. Det beror på vilken typ av program det är frågan om, vilka som ska använda det, samt i hur hög grad programmet är självinstruerande.

Programdokumentationen riktar sig till de personer som ska underhålla: Införa rättningar, ändringar eller vidareutveckla programmet.

Programdokumentationen för ett ABC80-program bör bestå av:

1. Programbeskivning.
2. Datastrukturer med variabelbeskrivning.
3. Programstruktur med operationslista.
4. Beskrivning av generella subrutiner.
5. Korsreferenstabell (vid stora program).
6. Programlista.

I programbeskrivningen talar vi i ord om vad programmet gör och vilka förutsättningar som gäller. Datastrukturer och programstruktur med operationslista kommer fram automatiskt i och med att vi använder JSP.

Vad som döljer sig bakom de andra begreppen ska avslöjas i följande avsnitt.

2.3.2 VARIABELBESKRIVNING

Vi kan dela upp de variabler vi använder i ett program i (minst) två kategorier. En del använder vi för att tillfälligt lagra data. Sådana variabler, vars värde blir kortlivat, kallar vi temporära variabler eller slaskvariabler. Enligt Ref-7 bör temporära variabler börja med bokstav P - U.

Den andra kategorin är sådana variabler som vi reserverar för vissa speciella data genom hela programmet. Det kan t ex röra sig om aktuellt postnummer vid läsning av en fil, senast inmatade sträng från tangentbordet etc. Sådana variabler kallar vi dedicerade variabler .

Dokumentationen bör innehålla en förteckning över dessa variabler och deras användning.

2.3.3 BESKRIVNING AV GENERELLA SUBRUTINER

En data- eller programstruktur ritar vi till ett visst djup. Det som återstår längst ut/ner i strukturen kallas löv. (Strukturen är en trädstruktur). I en datastruktur består löven av dataelement, i en programstruktur utgörs löven av operationer.

Till programstrukturen hör en lista som beskriver dessa operationer. Se exempel i appendix F. Somliga operationer förekommer på flera ställen och är av mera generell karaktär. Dessa kallar vi generella operationer eller generella subrutiner och beskriver dem för sig.

Denna beskrivning bör innehålla:

- En kort beskrivning av vad subrutinen gör.
- Invariabler.
- Utvariabler.
- Vilka andra subrutiner som anropas.

Beskrivningen kan, om vi har gott om minne, få stå som kommentarer i programmet.

I BASIC refereras en subrutin endast med radnummer (som ändrar sig vid renumrering). Vi bör därför ge varje subrutin ett namn i kommentarform som även anges vid anropet. Se exempel nedan.

Exempel 2.1 Generell subrutin
Raderna 500 till 620 tas endast med i programmet om plats finns i minnet.

```
.
.
130 GOSUB 630 : REM *Stora*
.
.
.
500 REM ------------------------
510 REM ¦ S T O R A
520 REM ------------------------
530 REM Små bokstäver omvandlas
540 REM till stora.
550 REM -
```



```
560 REM Invariabel
570 REM - B$   Textsträng
580 REM Utvariabel
590 REM - B$   Textsträng
600 REM Temp. variabel
610 REM - P
620 REM ------------------------
630 REM *Stora*
640 FOR I=1 TO LEN(B$)
650 P=ASC(RIGHT$(B$,I))
660 IF P<96 THEN 680
670 B$=LEFT$(B$,I-1)+CHR$(P AND 223)+RIGHT$(B$,I+1)
680 NEXT I
690 REM *Stora-end*
700 RETURN
```

2.3.4 KORSREFERENSTABELL

I stora program händer det lätt att man råkar använda en redan upptagen variabel till något helt annat. Det inträffar speciellt gärna när man gör tillägg till ett program som man kanske inte skrivit själv. Fel av detta slag är ofta besvärliga att hitta.

För att förebygga detta gör vi, samtidigt som vi skriver programmet, en korsreferenstabell. Vi låter varje variabel vi använder få en rad i tabellen och varje subrutin en kolumn. Se figur 2.10.
I korsningarna anger vi om, och hur, respektive variabel används i respektive subrutin.

I - Betyder invariabel.
U - Betyder utvariabel.
X - Betyder temporär variabel vars värde förändras.

Ett ensamt I betyder att subrutinen ifråga "läser av" värdet på den variabeln men ej ändrar dess värde. Den måste alltså ges värde före anropet.

Ett ensamt U betyder att subrutinen lämnar ett resultat i denna variabel. Utvariabelns värde före anropet av subrutinen är ovidkommande.

Både ett I och ett U slutligen betyder att variabeln i fråga både är in- och ut-variabel. Dess värde kan ändras av subrutinen.

Andra kombinationer förekommer ej.

En subrutin kan ju mycket väl anropa en annan subrutin som i sin tur anropar en subrutin osv. För att ta reda på vilka variabler som påverkas måste vi alltså även titta på de underliggande subrutinerna. För att underlätta detta anger vi längst ner i tabellen vilka andra subrutiner som anropas. Se figuren.

Man kan ha mycket stor nytta av en korsreferenstabell.

När vi behöver en ny variabel vet vi vilka som är upptagna och vilka som är lediga. Att införa onödigt många variabler i ett program slukar minne. Dessutom är alfabetet ju begränsat!

Subrutiner / Variabler

	Läsrad	Skrivrad	Lässymb	Nysymb	Fel	Cita
B				U		
D			X	IU		IU
D1$	U		IU	IU		IU
D2$		I	IU	IU		IU
E9	U	U			I	
K						X
Anrop				Cita		

Fig 2.10 Korsreferenstabell

2.4 Felbehandling — ON ERROR GOTO

Fel i programmet kan leda till programstopp och felutskrift, t ex "ERR 12 LINE 120". Flertalet av de felsituationer som uppkommer kan emellertid tas om hand av programmet självt.

Exempel: Felaktig inmatning från tangentbordet.
Här bör ett användarvänligt program tala om felet och fråga om.

Med BASIC-satsen ONERRORGOTO kan vi ta hand om sådana fel. Detta är dock inte helt riskfritt om man utformar sitt program fel. Vi ska studera detta i följande exempel (typ avskräckande).

Exempel 2.2 Så här får det inte se ut!

```
100 GOSUB 130 : REM * Subr1 *
110 END
120 ;
130 REM * Subr1 *
140 ONERRORGOTO 180
150 GOSUB 220 : REM * Läs *
160 PRINT "Läs gick bra"
170 GOTO 190
180 PRINT "Läs gick fel"
190 REM * Subr1-end *
200 RETURN
210 ;
220 REM * Läs *
230 READ A
240 REM * Läs-end *
250 RETURN
```

```
RUN
Läs gick fel
Läs gick bra

ABC80
```

Programmet sätter i SUBR1 ONERRORGOTO på så sätt att programflödet ska ta två olika vägar efter anropet av subrutinen LÄS. En om det inte uppstår fel. En annan om det blir fel.

Men vad händer vid fel (det finns inga DATA)?

Jo programmet tycks ha tagit båda vägarna på en gång! Förklaringen är att GOSUB till LÄS inte kom att åtföljas av något RETURN. BASIC-interpretatorn sparade återhoppsadressen (rad 160) vid anropet av LÄS och denna låg kvar då RETURN på rad 200 skulle tolkas.

Genom att först ge kommandot TRACE och sedan köra programmet igen kan vi se instruktionsföljden.

Vi måste därför ställa upp föjande regler för användning av ONERRORGOTO:

1. ONERRORGOTO får inte vara aktiv vid subrutinanrop (GOSUB).

2. ONERRORGOTO får inte vara aktiv vid återhopp från subrutin (vid RETURN).

ONERRORGOTO blir passiv (inget hopp) genom att sätta radnumret 0 (ONERRORGOTO 0).

Programmet skulle istället ha sett ut så här:

Exempel 2.3 Rätt!

```
100 GOSUB 130 : REM * Subr1 *
110 END
120 ;
130 REM * Subr1 *
140 GOSUB 220 : REM * Läs *
150 IF E9<>0 THEN 180
160 PRINT "Läs gick bra"
170 GOTO 190
180 PRINT "Läs gick fel"
190 REM * Subr1-end *
200 RETURN
210 ;
220 REM * Läs *
230 ONERRORGOTO 270
240 READ A
260 GOTO 280
270 E9=ERRCODE
280 ONERRORGOTO 0
285 REM * Läs-end *
290 RETURN

RUN
Läs gick fel

ABC80
```


KAPITEL 3

Dataelement

ABC80

3.1 Variabler i BASIC

En namngiven enhet data kallas dataelement. Dataelementen i BASIC är heltal, flyttal, strängar, vektorer och matriser.

Heltal, flyttal och strängar är grundelement, dvs. dataelement som vi (oftast) inte har anledning att sönderdela ytterligare. Vektorer och matriser kallas även indexerade variabler.

I ABC80-BASIC finns även instruktioner som gör det möjligt att läsa och skriva direkt i minnet. Enheten är då en BYTE (8 bitar), som är den minsta enheten vi har anledning att intressera oss för i ett BASIC-program.

Sektionerna 3.2 - 3.6 behandlar de variabeltyper vi kan använda oss av i BASIC. I sektionerna 3.7 och 3.8 tittar vi närmare på speciella användningar av strängar (ASCII-aritmetik) samt heltal (logiska uttryck).

3.1.1 KONVENTIONER

För att ett program skall vara lätt att förstå, både för andra och för den som en gång skrivit det, bör man följa vissa konventioner. Vi ska inte gå närmare in på dessa, det görs i boken "Programmering för dig och för mig". (Ref-7).

- Dimensionering av strängar, vektorer och matriser sker i början på programmet.

- Tilldelning av begynnelsevärden till variabler (initiering) följer därefter.

- Av bokstaven i variabelnamn bör framgå hur variabeln används:

  A - H : konstanter och indata.
  I - N : index och loopvariabler.
  P - Q : temporära variabler (slask).
  V - Z : resultatvariabler.

  Speciellt bör variabler på bokstaven O undvikas, då O och 0 (noll) ser likadana ut på ABC80-skärmen.

3.1.2 MINNESBEHOV OCH EXEKVERINGSTID

För varje variabel som förekommer i ett program reserveras utrymme i en tabell, dels för information om variabeln (namn, typ etc), dels för dess värde. Denna tabell kallas SYMBOLTABELL (Se även kap. 8). Olika typer av variabler tar upp olika stor plats i symboltabellen.

Även exekveringstiden beror på vilka variabeltyper vi använder i programmet. Heltalsoperationer går t ex fortare än flyttalsoperationer.

Vill vi ha ett snabbt program, ska vi använda heltal överallt där det är möjligt.

Exempel:

```
10 FOR I=1 TO 10000 : NEXT I
```

tar ungefär 10 s. medan

```
10 FOR I%=1% TO 10000% : NEXT I%
```

bara tar 2 s.

3.2 Heltal

För att ange att en variabel ska lagras som heltal sätter vi ett procenttecken (%) sist i variabelnamnet.

Värdet av en heltalsvariabel lagras i två byte, med tvåkomplement. Med två byte (16 bitar) kan ett binärtal i intervallet 0 till 65535 (0000H till 0FFFFH) skrivas. Tvåkomplement innebär att talområdet istället blir -32768 till +32767. Av den mest signifikanta biten framgår om talet är negativt - den är då 1. Genom att invertera alla bitpositioner (ettor blir nollor, nollor blir ettor) samt lägga till 1 får vi ett negativt tals absolutvärde.

Exempel.

Binärt	Hexadecimalt	Decimalt
0111111111111111	7FFF	+32767
0000000000000101	0005	+5
0000000000000000	0000	0
1111111111111111	FFFF	-1
1111111111111011	FFFB	-5
1000000000000001	8001	-32767
1000000000000000	8000	-32768

Vi måste givetvis tänka på det begränsade talområdet när vi använder heltal.

Låt oss prova följande på ABC80:

```
10 A%=70000
20 ;A%

ERR 7 LINE 10

ABC80
```

Vi får felindikering när vi försöker tilldela en heltalsvariabel ett värde större än 65535 (OFFFFH). Vi fortsätter:

```
10 A%=65535
RUN
-1

ABC80
```

A% = 65535 och A% = -1 ger samma resultat (-1) !
Vi inför:

```
15 A%=A%-32768
RUN
 32767

ABC80
```

Variabeln "slår runt" utan att vi får någon felindikering på detta. Vi ska därför endast använda heltalsvariabler där vi kan garantera att deras värde hamnar inom talområdet.

Vi måste särskilt se upp med FOR - loopar:

```
10 FOR I%=30000% TO 32000% STEP 1000%
20 PRINT I%;
30 NEXT I%

RUN
 30000 31000 32000-32536-31536-30536
-29536-28536-27536-26536-25536-24536
-23536  .....
```

Vad hände?
Jo, i en FOR - loop ökas först räknaren (I%), därefter testas på slutvillkoret. I det som skulle vara sista varvet blev 32000 + 1000 = 33000, som tolkas som -32536! Loopen fortsätter.

Heltalsvariabler tar å andra sidan liten plats och medger snabba program.

En heltalsvariabel tar upp 6 byte i symboltabellen.

3.3 Flyttal

Ett tal som framställs på formen

$$+/- \; M * 10^{E}$$

kallas flyttal. M benämns mantissa och E exponent.

I ABC80 består mantissan av 6 decimala siffror som lagras i tre byte. Exponenten + 128 lagras binärt i en byte.

Flyttalen får därmed en noggrannhet av 6 siffror med en exponent mellan -127 och +128.

Värdet av den lagrade mantissan ligger alltid i intervallet +/- 0.1 till +/- 0.999999 . Utskriftsrutinen presenterar flyttal på så sätt att mantissan ligger mellan +/- 1.0 och +/- 9.99999 .

Ett flyttal tar upp 9 byte i symboltabellen.

3.4 Strängar

Variabelnamnet för en sträng slutar alltid med strängtecknet ($). En sträng lagras som en följd av bytes för lagring av t ex text. Strängens maximala längd kan deklareras med en DIM-sats. Dimensioneras ej strängen reserveras plats för 80 tecken.

Vid stora program bör vi spara plats genom att dimensionera även strängar som vi vet behöver mindre än 80 tecken. Exempel:

```
50 DIM A$=1
```

Storleken kan även anges med en variabel:

```
40 C=5
50 DIM C$=C
```

Observera att stränglängden inte kan utökas under exekveringen.

DIM-satserna bör samlas i programmets början (Ref. 7).

En sträng tar upp 10 bytes i symboltabellen förutom deklarerad längd.

3.5 Byte

Med instruktionen PEEK kan vi läsa data var som helst i adressområdet, både i ROM- och RAM-minnet. Med instruktionen POKE kan vi lagra data i RAM-minnet.

De data vi önskar lagra kan utgöras av maskininstruktioner som vi sedan anropar med CALL, det kan vara variabelvärden som ska överföras mellan två program då CHAIN användes eller vi kan helt enkelt behöva lagra en tabell eller liknande i kompakt form.

En byte har som bekant talområdet 0 till 255.

Det utrymme vi i första hand bör utnyttja är 128 bytes från och med adress 65408 högst upp i minnet. Behövs ytterligare utrymme kan vi få detta genom att höja programgolvet (BOFA, Se appendix-D). Pekaren som talar om var programmet ska börja lagras - BOFA - finns på adress 65052 - 65053. Genom att öka denna innan vi läser in vårt program kommer programmet att lagras högre upp i minnet. Detta kan vi göra som ett kommando eller med ett litet program som sedan m h a CHAIN läser in det egentliga programmet. BOFA är normalt satt till 49152 (OC000H).


```
10 REM Sätt BOFA till 49408 (OC100H)
20 A=49408
30 POKE 65052,A,SWAP%(A)
40 PRINT "BOFA ändrad till:";PEEK(65053)*256+PEEK(65052)
50 CHAIN "PROGRAM"
```

Programmet "PROGRAM" kan nu disponera utrymmet 49152 till 49407 för PEEK och POKE.

Observera att ABC80 lagrar adresser och pekare som omfattar två byte med den minst signifikanta byten på den lägsta adressen. I exemplet ovan såg det ut så här:

```
65052 : 000H  )
65053 : 0C0H  ) BOFA = 0C000H
```

som vi ändrade till

```
65052 : 000H  )
65053 : 0C1H  ) BOFA = 0C100H
```

3.6 Vektorer och matriser

3.6.1 NAMN

Vektorer och matriser får namn enligt den typ av grundelement som vektorn eller matrisen består av. Ett eller två index pekar ut det element vi är intresserade av. Exempel: A(4), B$(2,3), C%(1%,4%).

3.6.2 DIMENSIONERING

Vi kan reservera plats i minnet med BASIC - satsen DIM. DIM står för "dimensionera". Om vi utelämnar DIM-satsen reserveras, för vektorer samma plats som om vi hade skrivit DIM (10), och för matriser DIM (10,10). Detta betyder att plats reserveras för 11 respektive 121 element ! Dvs. om vi skriver

```
10 A(3,2)=19
```

så motsvarar det

```
5 DIM A(10,10)
10 A(3,2)=19
```

Observera att index räknas från noll och uppåt. Detta innebär att matrisen som dimensionerats:

```
10 DIM C(1,1)
```

kommer att innehålla 4 element, nämligen C(0,0), C(0,1), C(1,0) och C(1,1).

Om vi ska dimensionera en vektor eller matris bestående av strängar, skriver vi t ex

```
10 DIM A$(9)=40
```

vilket reserverar plats för 10 strängar, vardera 40 tecken långa.

En vektor eller matris som en gång dimensionerats i ett program kan inte längre fram i programmet utökas. Däremot är det möjligt att minska dimensioneringen. Detta medför dock INTE att något minne frigörs. Det finns därför inte någon anledning att dimensionera om vektorer eller matriser.

Dimensionering med en variabel,

```
10 DIM A$(A%)=B%
```

kan däremot vara meningsfullt då vi enkelt vill kunna anpassa vårt program till olika behov.

3.6.3 MINNESBEHOV

Den plats som reserveras i symboltabellen för en vektor eller matris kan vi få ur dessa formler:

Vektorer: M = 10 + N * E

Matriser: M = 14 + N * E

där

M är minnesbehov i bytes.

N är antal element (11 respektive 121 för vektorer respektive matriser som ej har dimensionerats).

E är minnesbehovet för ett element:
2 för heltal,
5 för flyttal,
6 + dekl. längd för strängar.

3.7 Decimaltal i strängar — ASCII aritmetik

När vi behöver större noggrannhet än heltal och flyttal ger, kan vi använda oss av den s.k. ASCII-aritmetiken. Den fungerar på strängar som kan tolkas som ett tal. Strängen kan innehålla tecken, siffror och decimalpunkt, men inte exponent.

Med hjälp av funktionerna ADD$(A$,B$,N), SUB$(A$,B$,N), MUL$(A$,B$,N) och DIV$(A$,B$,N) kan vi utföra addition, subtraktion, multiplikation och division av numeriska strängar. Resultatet blir också en sträng med det numeriska värdet korrekt avrundat till N decimaler.

Exempel:

```
10 A$="1"
20 B$="7"
30 PRINT DIV$(A$,B$,12)
40 PRINT DIV$(A$,B$,10)
RUN
.142857142857
.1428571429

ABC80
```

Observera dock att resultatsträngen inte kan innehålla mer än 28 tecken med decimalpunkt och eventuellt minustecken inräknade, och därför får vi fel om resultatet blir för stort eller vi vill ha för många decimaler. Här bör noteras att beräkningar med ASCII-aritmetik tar väsentligt mycket längre tid än om det vore vanliga tal. Exempel:

```
10 G%=0% : G=0 : G$="0"
20 FOR I%=1% TO 500%
30 G%=G%+1%
40 NEXT I%
```

tar ca. 5 sekunder att köra. Om vi ändrar rad 30:

```
30 G=G+1
```

så tar det ca. 10 s. och om vi skriver

```
30 G$=ADD$(G$,"1",0)
```

så tar det ca. 60 s.

För jämförelse av två numeriska strängar kan vi inte använda de vanliga relationsoperatorerna. Exempel:

```
10 IF "-1"<"+1" THEN ; "mindre" ELSE ; "
större"
RUN
större
```

Strängen "-1" är alltså större än strängen "+1", och det duger ju inte om vi är intresserade av talen strängarna representerar. Därför finns funktionen COMP%(A$,B$) som jämför två numeriska strängar tolkade som tal. Värdet av COMP% är:

-1 om (talet i A$) < (talet i B$)

0 om de är lika

+1 om (talet i A$) > (talet i B$)

Vår sats ovan ändras till:

```
10 IF COMP%("-1","+1")=-1 THEN ; "mindre
" ELSE ; "större"
RUN
mindre
```

och nu stämmer det!

3.8 Logiska uttryck

Ett logiskt uttryck är ett uttryck som bara kan anta två värden; SANT eller FALSKT. T ex (A > 2) är ett logiskt uttryck som är SANT om A är större än 2 och FALSKT annars. I ABC80 representeras SANT med -1 och FALSKT med 0. Detta är rent numeriska värden och kan också användas som sådana.

En logisk variabel är en vanlig (tal-) variabel, som tilldelas värdet av ett logiskt uttryck. Det finns alltså ingen kontroll på att variabeln bara tilldelas något av värdena 0 el -1.

T ex

```
10 A = (B > 2)
```

är detsamma som

```
10 IF B > 2 THEN A = -1 ELSE A = 0
```

Då kan vi betrakta A som en logisk variabel, men om vi däremot skriver

```
20 A = -4 * (B > 2)
```

som är ekvivalent med

```
20 IF B>2 THEN A=4 ELSE A=0
```

ska vi INTE betrakta A som en logisk variabel. Visserligen fick A sitt värde från ett logiskt uttryck, men dessutom multiplicerade vi med en konstant (-4) så A:s möjliga värden är inte längre 0 el. -1.

Varför ska vi vara så hårda?

Jo, det är så, att IF-satsen anser att allt <>0 är SANT vilket inte är lika strängt som att SANT = -1. Titta noga på följande program!

```
10 FOR A=-1 TO 1
20 ; : ; "A="A
30 IF A THEN ; "A är sant"
40 IF NOT A THEN ; "NOT A är sant"
50 NEXT A
RUN

A= -1
A är sant
```

```
A= 0
NOT A är sant

A= 1
A är sant
NOT A är sant

ABC80
```

NOT betyder ju ICKE. Om A är SANN så är NOT A FALSK och vice versa. Men vad hände vid tredje försöket? Jo, A hade värdet 1 som inte är något logiskt värde. Visserligen är A sann för IF-satsen men NOT A är också sann! Därför ska vi alltid se till att det är regelrätta logiska värden vi testar på i IF-satser. Om vi ändrar raderna 20 och 30:

```
30 IF (A<>0) THEN ; "(A<>0) är sant"
40 IF NOT (A<>0) THEN ; "NOT (A<>0) är sant"
RUN

A= -1
(a<>0) är sant

A= 0
NOT (A<>0) är sant

A= 1
(A<>0) är sant

ABC80
```

Och nu är det inte dubbeltydigt längre.

Som vi såg tidigare kan vi använda logiska uttryck i beräkningar. Detta rekommenderas dock inte, eftersom programmen blir mer svårlästa och risken att göra fel blir större. Det finns dock ett fall då vi kan tillåta oss att använda det, och det är i användardefinierade funktioner. Dessa får ju bara bestå av en tilldelning, och med hjälp av de logiska uttrycken kan vi få in villkor i denna tilldelning. Exempel funktionen MIN(a,b) lämnar det minsta talet av a och b som resultat. Behöver vi den bara en enda gång kan vi skriva

```
1000 IF A<B THEN C=A ELSE C=B
```

men om vi behöver den på flera ställen är det bättre att definiera en funktion som sedan anropas:


```
1000 C=FNM1(A,B)
```

Vi måste då i början av vår BASIC-program definiera FNM1, och där är det viktigt att vi skriver ordentliga kommentarer som talar om vad funktionen gör, det syns ju inte på namnet.

Exempel:

```
10 REM - FNM1(P1,P2) ger det minsta
20 REM - värdet av talen P1 och P2.
30 REM - Fungerar på heltal och flyttal.
40 DEF FNM1(P1,P2)=-P1*(P1<P2)-P2*(P1>=P2)
```

Hur fungerar den? Vi har tre fall:

```
P1 < P2 : FNM1=-P1*(-1)-P2*(0) = P1
P1 = P2 : FNM1=-P1*(0)-P2*(-1) = P2
P1 > P2 : FNM1=-P1*(0)-P2*(-1) = P2
```

Och det stämmer ju.

KAPITEL 4

Filer

ABC80

4.1 Inledning

För att spara undan program och data för senare återanvändning utnyttjar vi filer.

Vi kan ha filer på kassettband eller diskett (flexskiva), och användningen blir lite olika beroende på vilket av dessa media vi utnyttjar.

BEGREPPSDEFINITIONER

FIL
: En följd av poster (exempel bilregister).

POST
: En datauppsättning om en individ (exempel information om en bil).

FÄLT
: Ett dataelement, del av post (exempel bilens färg).

NYCKEL
: Ett fält som är unikt för individen (exempel bilnummer).

SEKVENTIELL FIL
: Fil med poster lagrade efter varandra i följd, med separationstecken (RETURN) mellan varje post. En enskild post kan bara nås genom att man läser förbi alla poster som står före (exempel .BAS programfiler).

DIREKTFIL
: Fil där vilken post som helst kan läsas direkt, utan att man behöver läsa förbi andra poster.

4.2 Organisation

Alla filer, både på kassett och diskett, är uppdelade i block om vardera 256 bytes.

Vid skrivning på en fil lagras först de data vi skriver i en buffert i minnet (CASBUF1-2, DOSBUF0-7), som omfattar 256 byte. Varje fil vi har öppnat i vårt program får en egen buffert. När den är full överförs hela bufferten på en gång till kassetten/disketten.

Samma sak sker när vi begär inmatning från en fil. Då kommer ett helt block att överföras till bufferten, och sedan tas data ur denna tills den är tom, varvid ett nytt block överförs, osv.

Av de 256 byten i ett block behöver systemet 3 till identifikation (filnummer och blocknummer), så det finns 253 databyte över i varje block för användaren.

4.3 Kassetten

En fil på kassetten består av tre distinkta delar; först ett block med filens namn, sedan ett varierande antal datablock och sist ett slutblock.

Om vi skriver ut data på kassetten så fort det går, lagras blocken helt utan mellanrum, men om det går längre tid mellan skrivningarna blir det tysta pauser mellan blocken (eftersom kassetten går hela tiden filen är öppen). Dessa kan bli så långa att vi får fel (ERR 42, enheten ej klar) när vi försöker läsa filen igen.

4.3.1 SKRIVNING OCH LÄSNING AV KASSETTFILER

Här följer ett kort exempel på hur man skapar en kassettfil, skriver på den och sedan läser filen från kassetten.

```
10 REM Skapa filen
20 PRINT "Ställ bandspelaren på inspelning."
30 PRINT "Tryck RETURN när du är klar!"
40 GET A$ : REM Vänta på tangent
50 PREPARE "CAS:FOO.DAT" ASFILE 1
```

```
60 REM Skriv på filen
70 PRINT #1,"DATAFIL NR 1"
80 FOR I=1 TO 10
90 PRINT #1,I
100 NEXT I
110 REM Stäng filen
120 CLOSE 1
130 PRINT
140 PRINT "Spola tillbaka kassetten och sätt den"
150 PRINT "på avspelning."
160 PRINT "Tryck RETURN när du är klar!"
170 GET A$ : REM Vänta på RETURN
180 REM Öppna filen
190 OPEN "CAS:FOO.DAT" ASFILE 1
200 REM Läs data från filen
210 INPUT #1,A$ : PRINT A$ : REM Skriv på skärmen
220 FOR I=1 TO 10
230 INPUT #1,Z : PRINT Z;
240 NEXT I
250 CLOSE 1 : END
```

Observera att vid INPUT packas det inlästa ihop - alla mellanslag tas bort - medan vid INPUTLINE läses strängen in precis som den står på filen.

Exempel om det på filen står: "DATAFIL NR 1"
och vi gör

```
10 INPUT #F,A$
```

så får A$ värdet "DATAFILNR1", men om vi gör

```
10 INPUTLINE #F,A$
```

så får A$ värdet "DATAFIL NR 1"+CHR$(13)+CHR$(10)

Nu finns mellanslagen kvar, men vi får dessutom med tecknen för ett RETURN och ett LINE FEED sist i strängen.

En sak att lägga på minnet är, att när vi läser en sträng från en fil så kommer en hel rad (dvs upp till och med ett RETURN) att läsas in till strängen. Om vi inte tänker oss för när vi skriver ut våra data på filen kan resultatet bli ett helt annat än vi väntar oss!

Om vi t ex har gjort följande i vårt program

```
1000 PREPARE "CAS:FOO.TXT" ASFILE 1
1010 A$="Hejsan"
1020 B$="Hoppsan"
1030 PRINT #1,A$;B$
1040 CLOSE 1 : END
```

så har vi ju skrivit ut två strängar på filen, men eftersom det INTE står tecknet RETURN mellan dem, kan inte ABC80 veta det när vi läser in texten igen!

Dvs

```
1000 OPEN "CAS:FOO.TXT" ASFILE 1
1010 INPUTLINE #1,X$
1020 PRINT X$
1030 END
```

ger resultatet:

```
HejsanHoppsan
```

när vi kör det. Vi måste alltså komma ihåg att göra ny rad i filen för alla data som vi vill kunna läsa in likadant som vi skrev dem.

De fel som kan uppstå när vi läser från kassettband är:

1. ERR 21 - hittar ej filen. Oftast har det varit för lång tystnad före filen på bandet. Spola fram en liten bit (inte så långt att man hör "ljudet av bitarna") och försök igen.

2. ERR 34 - filen slut.

3. ERR 35 - checksummafel. Filen går inte att läsa. Spola tillbaks kassetten och försök igen.

4. ERR 37 - felaktigt recordformat. Som ovan.

5. ERR 42 - enheten ej klar. Detta beror antingen på att bandspelaren inte är igångsatt, eller att det är för långa pauser mellan blocken på filen.

Både för tillförlitlighets och bekvämlighets skull bör vi bara

använda en sida av kassettbanden. Även på en sida av en C60-kassett ryms åtminstone lika mycket som på en och en halv diskett!

4.3.2 STORA DATAFILER

Att behandla data som finns på kassett är enkelt om filen är så liten att vi kan läsa in alltihop till minnet med en gång, eller om vi kan behandla data minst lika fort som de kommer från kassetten.

Men om så inte är fallet får vi problem. Data kommer in från kassetten fortare än vi förbrukar dem - när bägge kassettbuffertarna är fulla slutar ABC80 ta emot data från bandet, men det snurrar lika fullt vidare!

Lösningen på detta är att vi stoppar kassettmotorn mellan varje datablock, och när ett block är förbrukat startar vi motorn igen och läser nästa.

Detta går dock bara att göra om vi har specialpreparerade filer med lite extra pauser mellan blocken - motorn behöver ca. 1.5 sek. för att komma upp i varv.

För att kunna göra denna extra paus mellan varje utspelat block, måste vi veta exakt när ett block har spelats ut på kassetten.

Exempel 4.2

```
100 DO$="CAS:FOO.TXT" : DO%=1
110 GOSUB 320 : REM Skapa filen
.
.
150 D1$="Det vi vill skriva"
160 GOSUB 380 : REM Skriv på filen
.
.
200 GOSUB 500 : REM Stäng filen
.
.
300 END
310 REM ------------------------------------------------
320 REM Skapa kassettfil
330 REM
340 PREPARE DO$ ASFILE DO%
```



```
350 D1%=PEEK(65021) : REM Blocknummer
360 RETURN
370 REM ------------------------------------------------
380 REM Skriv på filen
390 REM
400 PRINT #DO%,D1$
410 REM
420 REM Testa om dags för paus
430 REM
440 IF D1%<>PEEK(65021) THEN GOSUB 440
450 RETURN
460 REM ------------------------------------------------
470 REM Paus mellan blocken
480 REM
490 FOR Z=0 TO 1500 : NEXT Z
500 D1%=PEEK(65021) : REM Nytt blocknummer
510 RETURN
520 REM ------------------------------------------------
530 REM Stäng filen
540 REM
550 REM Fyll ut med NUL-tecken tills ett
560 REM block har spelats ut.
570 IF D1%<>PEEK(65021) THEN 560
580 PRINT #DO%,CHR$(0); : GOTO 540
590 GOSUB 440 : REM Paus
600 CLOSE DO%
610 RETURN
```

För detta utnyttjar vi en av ABC80's systemvariabler. I cell 65021 i minnet står numret på det senast utskrivna blocket (255 om inget block alls är skrivet).

Genom att testa denna cell varje gång vi har skrivit på kassetten kan vi upptäcka när det är dags att göra en paus.

Vi kan också råka ut för motsatsen: Vi har ett program som tar mycket tid på sig för att generera data. Dessa skrivs då ut så sällan att det blir för långa pauser mellan blocken om kassetten får gå hela tiden. Då kommer vi att få ERR 42 - enheten ej klar - när vi försöker läsa filen.

Detta löser vi genom att starta och stoppa motorn även när vi skriver ut data på filen. För att detta ska fungera måste vi samla ihop (ungefär) ett block i en sträng, starta motorn och låta den komma upp i varv, skriva ut alla värdena i strängen och sedan stoppa motorn igen.

Med hjälp av subrutinpaketet i appendix G kan vi göra detta.

Subrutinerna är skrivna så, att varje filoperation byts ut mot ett subrutinanrop.

Utan subrutiner	Med subrutiner
PREPARE "FOO" ASFILE 1	DO$="FOO" : DO%=1% GOSUB 1350 : REM *Prepare*
PRINT #1,A;B%;C$	D1$=NUM$(A)+NUM$(B%)+C$+CHR$(13) DO%=1% GOSUB 1110 : REM *Print*
CLOSE 1	D=%=1% : GOSUB 1760 : REM *Close*

Efter varje subrutinanrop bör vi också testa om det blev fel:

```
IF E9<>0 THEN ???? : REM Vår felrutin
```

Dessa subrutiner ger även lagom långa pauser mellan blocken, så att filerna ska kunna läsas med start och stopp av motorn.

4.3.3 LÄSNING MED START OCH STOPP AV MOTORN

Nu har vi en fil med lämpliga pauser - hur ska vi bära oss åt för att läsa den?

Eftersom ABC80 läser ett helt block i taget från kassetten, räcker det med att vi startar motorn före varje INPUT-sats, och stoppar den direkt efter.

Om data finns i kassettbufferten går läsningen så fort, att motorreläet bara klickar till (kassettbandet hinner inte röra sig), men om bufferten är tömd kommer ett helt block att läsas in innan vi kommer till satsen efter INPUT-satsen, och stoppar motorn igen.

Vi bör också ta hand om eventuella fel , t.ex ERR 42 - som kan uppstå om pauserna på kassettbandet är för långa - och ERR 34 som innebär att filen är slut.

Även för detta finns subrutiner i appendix G. Precis som vid rutinerna för skrivning byts alla filoperationer ut mot subrutinanrop.

Utan subrutiner	Med subrutiner
OPEN "CAS:" ASFILE 1	DO$="CAS:" : DO%=1% GOSUB 2160 : REM *Open*
INPUTLINE #1,A$	DO%=1% GOSUB 2350 : REM *Input1* A$=D1$
INPUT #1,A	DO%=1% GOSUB 2350 : REM *Input1* A=VAL(LEFT$(D1$,LEN(D1$)-2)
CLOSE 1	CLOSE 1

Även här bör vi testa E9 efter varje subrutinanrop. Om E9 är skild från noll, så har ett fel uppstått i subrutinen.

4.4 Disketten

Disketten är uppdelad i 40 spår om vardera 8 sektorer. Varje sektor rymmer ett datablock (256 byte). Detta benämns även (fysisk) RECORD.

Spår 0 - 2 är upptagna av systeminformation, men resten kan vi använda till våra filer (program eller data).

Tack vare att alla filoperationer sköts av ett färdigt program - DOS-et (Disk-Operativ-Systemet) - behöver vi inte bekymra oss om spår och sektorer, och var filer egentligen hamnar, utan vi behöver bara öppna en fil, så kan vi läsa och skriva fritt.

En fil på disketten består - ur användarens synvinkel - av ett antal datablock, konsekutivt numrerade från 0 och uppåt. Vi kan antingen läsa/skriva rent sekventiellt, som på kassetten, eller direkt specificera vilket block i filen som ska läsas/skrivas (direktfiler).

4.4.1 SKRIVNING OCH LÄSNING AV DISKETTFILER

Filer skapas och öppnas på disketten precis som på kassetten, med PREPARE och OPEN.

Det är mycket viktigt att vårt program gör CLOSE när det har skrivit färdigt en diskett-fil. Om vi glömmer det (eller tar disketten ur driven innan det är gjort), är chanserna mycket stora att vi sitter med en kvaddad diskett!

Om vi är osäkra på om programmet har stängt filen eller inte kan vi efter körningen ge kommandot CLEAR, som stänger alla filer och nollställer alla variabler.

Med hjälp av CALL kan vi direkt läsa eller skriva ett bestämt block i en fil. Variabeln QO$ är reserverad för detta ändamål.

LÄSNING

```
Z=CALL(28666,F)+CALL(28668,B)
```

Där F är filnumret och B numret på blocket vi vill läsa. Resultatet (alla 253 databyten i blocket) hamnar i QO$, och om blocket innehåller flera poster får vi själva dela upp QO$ i dessa.

SKRIVNING

```
CALL(28666,F)
QO$="de data vi ska skriva"
Z=CALL(28670,B)
```

Innehållet i QO$ kommer att skrivas över det som tidigare stod i block nummer B.

Fel som kan uppstå vid läsning/skrivning på detta sätt är:

1. ERR 37 - felaktigt recordformat. Kan bero på att vi har glömt att preparera filen (se nedan).
2. ERR 38 - större blocknummer än det finns block i filen.
3. ERR 42 - ingen skiva i driven, eller luckan är öppen.
4. ERR 43 - skivan är skrivskyddad.

En direktfil måste prepareras innan vi kan läsa och skriva hur som helst på den. Vi måste bestämma oss för hur många block filen ska bestå av, och skriva ut något lämpligt på varje block (t ex mellanslag).

Exempel 4.3 Direktfilspreparering

```
100 ; CHR$(12)"Direktfilspreparering"
110 DIM QO$=253
120 ; : ; "Filnamn"; : INPUT DO$
130 ONERRORGOTO 270
140 REM
150 REM Gör först ett försök att öppna filen
160 REM för att se om den redan finns, och
170 REM avbryt i så fall.
180 REM
190 OPEN DO$ ASFILE 1 : CLOSE 1
200 REM
210 REM Det gick bra att öppna. Filen
220 REM finns. AVBRYT!
230 REM
240 ; : ; "Filen finns redan!"
250 GOTO 440
260 REM
270 REM Filen fanns inte. Skapa den.
280 REM
290 PREPARE DO$ ASFILE 1
300 ONERRORGOTO 300
310 ; : ; "Hur många block (1-255)";
320 INPUT B
330 IF B<1 OR B>255 THEN 310
340 ONERRORGOTO 400
350 FOR I=0 TO B-1
360 Z=CALL(28666,1) : QO$=SPACE$(253)
370 Z=CALL(28670,I) : REM Skriv block
380 NEXT I
390 ; : ; "Klart" : GOTO 420
400 ; : ; "Fel:"ERRCODE" !"
410 ; : ; "Vi avbryter!"
420 ; : ; I" Block skapade."
430 CLOSE 1
440 END
```

För att spara plats på filerna utför ABC80 packning av mellanslag vid PRINT, och motsvarande UPPACKNING vid INPUTLINE (ej vid INPUT).

Detta går till så, att om vi skriver mer än två mellanslag i följd på en fil, byts detta ut mot tecknet TAB (CHR$(9)) följt av en byte som talar om hur många mellanslag det ska vara. Omvänt när vi läser: tecknet TAB plus nästa byte omvandlas till ett antal mellanslag.

Detta gäller bara när vi skriver/läser med PRINT/INPUTLINE, inte när vi använder CALL för direktåtkomst. Dvs om vi skriver på en fil med PRINT, och sedan läser med Z=CALL(...), blir teckenföljden TAB <n> INTE uppackad.

Exempel 4.4

```
10 DIM QO$=253
20 PREPARE "FOO.TXT" ASFILE 1
30 PRINT #1,SPACE$(253) : CLOSE 1
40 REM
50 OPEN "FOO.TXT" ASFILE 1
60 QO$=""
70 Z=CALL(28666,1)+CALL(28668,0)
80 FOR I=1 TO 4
90 ; ASC(QO$); : QO$=RIGHT$(QO$,2)
100 NEXT I
110 CLOSE 1
120 END


RUN
 9 253 13 3

ABC80
```

Här ser vi att den lästa variabeln - QO$ - inte består av 253 mellanslag utan av TAB (CHR$(9)) och talet 253. Därefter följer ett RETURN (CHR$(13)) och ett ETX (CHR$(3)) som betyder slut på filen.

4.4.2 SUBRUTINER FÖR DIREKTFILER

När vi använder direktfiler, är det ju oftast inte så, att våra poster är precis 253 byte långa. Om de är kortare än 127 byte kan vi packa ihop flera poster i varje block på filen.

I appendix H finns ett subrutinpaket som administrerar detta. Vi behöver bara ange ett filnummer, DO%, och ett postnummer, D2%(DO%),

så kommer rätt block att läsas in, och den valda posten, D1$, att läsas eller skrivas i blocket.

Med subrutinpaketet i appendix-H kan vi arbeta med flera filer samtidigt. Med hjälp av en filbeskrivning i form av DATA-satser kan vi ange namn,postlängd och fillängd för varje fil.

En direktfil måste, som vi tidigare sagt, först prepareras. Vi gör detta genom att fylla blocken med nollor.

Om något fel skulle uppstå i subrutinerna sätts variabeln E9 till värdet av ERRCODE, och vi kan alltså testa på detta efter subrutinanropet.

Exempel på användning av subrutinerna:

```
DEFINIERA FILER
                        DATA 2
                        DATA KUNDREG.DAT,84,100
                        DATA ARTREG1.DAT,42,200

                        RESTORE <>
                        GOSUB 20400 : REM *Filinit*

PREPARERA FIL
                        DO%=1% : REM fil-1
                        GOSUB 20660 : REM *Filprep*

ÖPPNA FIL
                        DO%=1% : REM fil-1
                        GOSUB 20890 : REM *Öppna*

LÄS POST
                        DO%=1% : REM fil-1
                        D2%(DO%)=13% : REM Post-13
                        GOSUB 21610 : REM *Läspost*

SKRIV POST
                        DO%=1% : REM Fil-1
                        D2%(DO%)=13% : REM Post-13
                        D1$="De data vi vill skriva ... "
                        GOSUB 21300 : REM *Skrivpost*

STÄNG FIL
                        D2%(DO%)=13% : REM Post-13
                        GOSUB 21100 : REM *Stäng*
```

4.5 Sökning

I de flesta fall då vi använder oss av filer, är vi också intresserade av att snabbt kunna hitta en utvald post.

Beroende på om vi har en sekventiell fil eller en direktfil blir sökmetoderna lite olika.

4.5.1 SÖKNING I SEKVENTIELLA FILER

Här finns bara en metod: Vi läser från början av filen och jämför varje post med den vi söker tills vi hittar den (eller filen är slut). Om vi söker en post som inte finns i filen måste vi således läsa igenom hela filen för att vara helt säkra på att den sökta posten inte finns.

Om man oftast söker på samma fält i en post - t ex bilnummer i ett bilregister - lönar det sig att ha filen sorterad efter detta fält. Tack vare sorteringen behöver vi inte längre läsa igenom hela filen för att vara säkra på att en post inte finns. Det räcker med att läsa så långt att posten i filen är större än den sökta.

Exempel liten fil med telefonnummer:

ANDERS	11111
BERTIL	12012
CESAR	8080
DAVID	301
ÖRJAN	22222

Om vi vill ha DAVID´s telefonnummer söker vi från början av filen och jämför namnen tills vi hittar rätt.

Men, om vi söker BOSSE i filen, så finns han inte där! Detta kan vi vara säkra på så fort vi kommit till CESAR vid sökningen. BOSSE kan ju inte komma efter CESAR när namnen är sorterade i bokstavsordning.

Exempel på en osorterad sekventiell fil, sökning och ändring av data i den är programmet MEDLEM som finns med på FLEXDEMO-skivan.

4.5.2 SÖKNING I DIREKTFILER

På direktfiler kan vi utnyttja flera metoder för sökning av data.

1. Vi kan söka sekventiellt genom posterna, precis som på en sekventiell fil.

2. Vår söknyckel kan översättas (transformeras) till ett postnummer, varpå vi direkt kan läsa posten. Exempel på detta är programmet LAGER på FLEXDEMO-disketten, där (varunummer-1000) ger postnumret.

3. I en sorterad fil kan vi använda oss av binärsökning. Detta går ut på att vi jämför vår söknyckel med en post mitt i filen, och om den sökta posten är större "behåller" vi den översta halvan av filen, annars den nedersta. Sedan upprepas förfarandet med den halva vi behöll: vi jämför den sökta posten med den mittersta, och behåller rätt halva osv. tills vi har hittat posten eller - när det bara finns en post kvar - vi är säkra på att den sökta posten inte finns i filen.

Vi tar ett exempel på binärsökning:

En liten fil med 24 poster (där nyckeln är en bokstav).

Sätt

S = nyckel till den post vi söker (söknyckel).
L = första posten i filen (=0)
H = sista posten (=23)
M = mittenposten

Säg att vi söker posten med nyckel "t".

1. M (mittenelementet) sätts till post nummer L+(H-L)/2 = 0+(23-0)/2 = 11 (vi bryr oss inte om decimalerna).

```
L                       M                       H
a b c d e f g h j k l m n o p r s t u v w å ä ö
                                    S
```

2. S är större än post M:s nyckel ("t" > "n") så vi behåller den översta halvan, dvs L sätts lika med M (11). Det nya värdet på M blir 11+(23-11)/2 = 17.

```
L           M           H
n o p r s t u v w x å ä ö
          S
```

3. S är mindre än M:s nyckel. Vi behåller den nedersta halvan (H sätts lika med M). Nytt M: 11+(17-11)/2 = 14

```
L     M     H
n o p r s t u
          S
```

4. S är större än M:s nyckel. L sätts till M. M = 14+(17-14)/2 = 15

```
L M   H
r s t u
    S
```

5. S är större än M:s nyckel. L sätts till M. M = 15+(17-15)/2 = 16

```
L M H
s t u
  S
```

Vi har hittat posten! Det behövdes bara 5 jämförelser innan vi hittade den. Om vi skulle söka sekventiellt hade vi behövt 17 stycken jämförelser!

Allmänt gäller - om vi har N st poster i filen - att vi måste göra i medeltal 2log(N) jämförelser vid binärsökning och (N/2) jämförelser vid sekventiell sökning.

Dvs så fort filen innehåller mer än 4 poster blir det färre jämförelser med binärsökningsmetoden.

Vi utgår då från, att vi alltid söker på det nyckelfält filen är sorterad efter (t ex bilnummer).

Om vi däremot vill söka på något annat fält (t ex bilar av märket

SALVO) är inte filen sorterad med avseende på detta fält, och då återstår ingenting annat än den sekventiella sökningen.

4.5.3 INDEXSEKVENTIELLA FILER

En väg att få snabbare åtkomst i medelstora direktfiler är att använda sig av en nyckeltabell.

En nyckeltabell innehåller nycklarna till alla posterna, och information om posternas läge i filen.

Denna tabell blir mycket mindre än hela datafilen, och kan finnas i ABC80´s minne (exempelvis i en matris). Sökning sker nu i tabellen, vilket går mycket fortare än på filen, och först när vi har hittat posten behöver vi läsa från filen. Sökningen i tabellen kan ske sekventiellt eller med hjälp av t ex binärsökning om tabellen är sorterad.

Exempel en texteditor där radnummer är nyckel:

Sökt rad: 21

```
Radnummertabell                Datafil

Radnr     Postnr
---------------              -------------------------
   10         1              Detta är rad nummer 10.
   20         2              Och detta är nummer 20.
 * 21         5 --           Här kommer 30.
   30         3   |          Rad 40.
   40         4    --> Och först här är rad 21!
---------------              -------------------------
```

Nyckeltabellen innehåller alltså radnumret (nyckeln) och vilket postnummer raden har i filen.

Men - har vi större datafiler går inte en fullständig nyckeltabell in i minnet. Då kan vi i stället tänka oss filen uppdelad i logiska sektioner. I ett bilregister t.ex kan datafilen delas upp i: Alla bilnummer som börjar på A, alla som börjar på B, alla på C etc.

Vi använder oss nu av en tabell som innehåller det minsta och största nyckelvärdet för varje sektion, och var i filen sektionen börjar (filen måste vara sorterad). Sökning sker först i tabellen, där vi får reda på vilken sektion vi ska leta vidare i. Inom denna sektion kan vi söka på valfritt sätt (sekventiellt, binärsökning etc.) för att hitta posten.

Vi tar som exempel att vi vill göra om telefonnummerregistret från 4.5.1. I indextabellen lägger vi då in första och sista namn inom varje sektion, och postnumret där sektionen börjar (var den slutar får vi ju från nästa sektions början).

Sökt person: GUSTAV

Indextabell

Från	Till	Postnr
ABRAHAM	CECILIA	1
CHRISTER	FREDRIK	11
* FÄRDINAD	JOHAN	17
JUSTUS	NISSE	22
MARTIN	ÖRJAN	31

Sektion av datafilen

17	FÄRDINAND	1111
* 18	GUSTAV	2020
19	GÖRAN	1212
20	HELMER	1201
21	JOHAN	3210

Denna filorganisation kallas indexsekventiell, och tabellen kallas indextabell.

4.5.4 SORTERING

Vi har i de tidigare avsnitten flera gånger nämnt att det är fördelaktigt att ha en fil sorterad (i t ex bokstavsordning). Frågan är bara: Hur gör vi detta?

Enklast är det förstås om filen från början skrivs sorterad, och nya poster skrivs in på rätt ställe med en gång.

Det enda vi behöver då, är en möjlighet att jämföra strängar i valfri kollationeringsordning (sorteringsordning), t ex ett korrekt svenskt alfabet (é efter e, ü efter y; å, ä, ö i rätt ordning och ingen skillnad mellan stora och små bokstäver).

Direkt jämförelse mellan strängar kan vi inte använda, eftersom den sker i enlighet med ASCII-tabellen, där tecknen inte ligger riktigt i den ordning vi vill ha dem.

I stället skaffar vi oss en egen "tabell" i en sträng (kollationeringssträng), där tecknen ligger i den ordning vi vill ha dem (subrutinen INIT SVENSK nedan).

För att jämföra två strängar får vi sedan gå igenom dem tecken för tecken och jämföra dessas position i kollationeringssträngen (subrutinen JÄMFÖR, exempel 4.5).

Är tecknen olika sätts variabeln S% till 1 eller 2 beroende på vilket tecken som är "störst", och jämförelsen avbryts. Är de lika fortsätter jämförelsen med nästa tecken tills någon sträng är slut, eller olikhet uppnåtts. Om strängarna är helt lika sätts variabeln till 0.

Exempel 4.5 Subrutin för jämförelse med valfri kollationeringsordning.

```
1000 REM ------------------------------
1010 REM Subrutin för jämförelse av två strängar,
1020 REM sorteringsordning ges av S$
1030 REM
1040 REM Ex :   S$="AaBbCc....ÅåÄäÖö0...9"
1050 REM
1060 REM
1070 REM Invariabler :
1080 REM S1$   Sträng som ska jämförass
1090 REM S2$        -"-
1100 REM
1110 REM Utvariabel :
1120 REM S%    = 0 ; S1$=S2$
1130 REM = 1 ; S1$>S2$
1140 REM = 2 ; S1$<S2$
1150 REM
1160 REM Lokala variabler :
1170 REM S1%   Längden av S1$
1180 REM S2%   Längden av S2$
1190 REM S3%   Aktuell strängposition
1200 REM S3$   Aktuellt tecken i S1$
1210 REM S4$   Aktuellt tecken i S2$
1220 REM
1230 REM ------------------------------
1240 REM *Jämför*
1250 IF S1$=S2$ THEN S%=0% : GOTO 1380
1260 S1%=LEN(S1$) : S2%=LEN(S2$)
1270 S3%=0%
```

```
1280 S3%=S3%+1%
1290 REM Är S1$ slut?
1300 IF S3%>S1% THEN S%=2% : GOTO 1380
1310 REM Är S2$ slut?
1320 IF S3%>S2% THEN S%=1% : GOTO 1380
1330 S3$=MID$(S1$,S3%,1%) : S4$=MID$(S2$,S3%,1%)
1340 REM Är tecknen lika?
1350 IF S3$=S4$ THEN 1280
1360 REM Jämför kollationeringsordning
1370 S%=(INSTR(1%,S$,S3$)>INSTR(1%,S$,S4$))+2%
1380 REM *Jämför-end*
1390 RETURN
1400 REM
1410 REM ------------------------------
1420 REM Initiering av S$ till en teckenföljd
1430 REM som anpassar sig till det svenska
1440 REM alfabetet.
1450 REM
1460 REM ------------------------------
1470 REM *Initsvensk*
1480 DIM S$=128,S3$=1,S4$=1
1490 S$=""
1500 FOR S%=0% TO 63%
1510 S$=S$+CHR$(S%)
1520 NEXT S%
1530 S$=S$+"AaBbCcDdEeÉé"
1540 FOR S%=70% TO 89%
1550 S$=S$+CHR$(S%)+CHR$(S%+32)
1560 NEXT S%
1570 S$=S$+"ÜüZzÅåÄäÖö"
1580 REM *Initsvensk-end*
1590 RETURN
```


KAPITEL 5

Interaktion människa–dator

ABC80

5.1 Olika tillämpningar — olika krav

Utformnigen av samspelet mellan användare och dator är kanske det viktigaste att tänka på när man konstruerar ett program.
Olika tillämpningar ställer olika krav. Ibland behövs speciell hårdvara i form av digitaliseringsbord, plotter, speciella skrivare och läsare etc. En enkel inmatningsenhet som utvidgar tangentbordets möjligheter går vi igenom i kapitel 7.
I detta kapitel ska vi begränsa oss till den dialog som vi kan föra med datorn via tangentbord och bildskärm samt ljudgenerator.

Vi kan som exempel titta på fyra olika tillämpningsområden:

- Beräkningsprogram
- Textbehandlingssystem
- Datainsamling, registrering
- Övervakning, styrning

Beräkningsprogram kräver ofta ett antal parametrar som indata. Enklast är då att låta programmet prompta (uppmana) genom att skriva namnet på variabeln i fråga.

```
110 PRINT "X" : INPUT X
```

Programmet förväntar sig ett värde följt av RETURN.

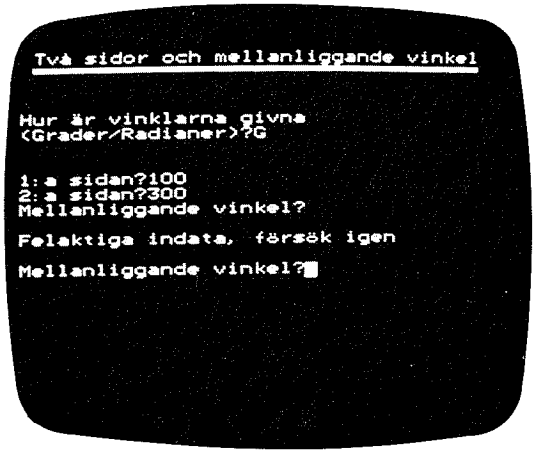


Fig 5.1 Från Matematik Pak 1.

Resultat kan ofta presenteras grafiskt.

Mera komplicerat kan det vara t ex vid textbehandling/editering. Här krävs flera funktionstangenter för att manipulera texten. På ABC80´s tangentbord finns -> och <- . För andra funktioner kan vi använda CTRL i kombination med andra tangenter. Inmatningsrutinen får här skrivas i BASIC och använda GET-instruktionen.

Datainsamling, typ registrering av verifikationer, kräver också sin speciella behandling. Här rör det sig ofta om stora datamängder, men med sins emellan likadana uppgifter.

Ett användarvänligt sätt att utforma sådana rutiner är att visa ett formulär med ledtexter på skärmen och låta operatören fylla i detta. Kontroll av indata är här extra viktig.

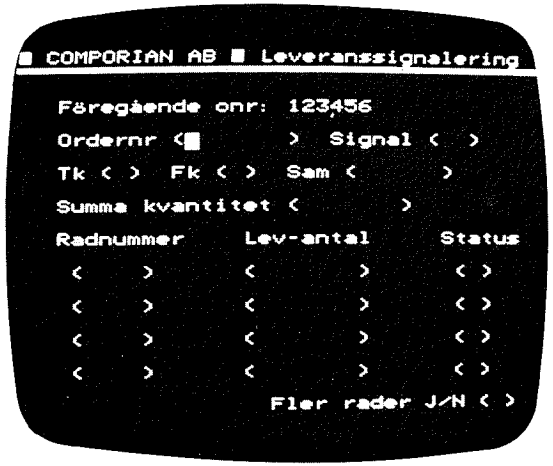


Fig 5.2 Dataregistrering med formulärteknik.

Vid övervakning/styrning slutligen ställs krav på att ABC80 ska ge en god överblick av t ex tillståndet hos en process och visa olika typer av larm. Detta kan ge oss anledning att använda såväl grafik, blinkande text som ljud.

Om ett program uppfattas som bra eller dåligt beror ofta på hur just dialogen med användaren fungerar, att indata kontrolleras, att adekvata meddelanden ges m m.

Den största delen av ett användarvänligt program kommer därför att gå åt till dialogen med människan.

Efter att ha stiftat närmare bekantskap med tangentbordet och bildskärmen ska vi gå igenom hur denna dialog kan utformas.

5.2 Tangentbordet

5.2.1 LAYOUT OCH KODER

Tangentbordet i ABC80 ser i stort sett ut som tangentbordet på en skrivmaskin. Tre tangenter avger inte något tecken utan bestämmer vilken uppsättning koder de andra tangenterna ska generera:

SHIFT — Ger tillsammans med bokstavstangenter versaler samt den övre symbolen av dubbeltecknade tangenter.

CTRL — Ger tillsammans med vissa andra tangenter koder som motsvarar kontrolltecken i ascii-tabellen.

UPPER CASE — Låser tangentbordets bokstavstangenter till versaler då tangenten lyser. SHIFT kan användas som vanligt.

Tangentbordslayout och koder framgår av appendix-C.

5.2.2 INMATNING MED INPUT, INPUTLINE OCH GET

Vi kan i ett BASIC-program begära inmatning av data med INPUT, INPUTLINE och GET. I föregående kapitel om filer behandlade vi instruktionerna INPUT och INPUTLINE. Observera att INPUT #0 och INPUTLINE #0 förväntar sig inmatning från tangentbordet. Tangentbordet kan även ses som en fil med speciella egenskaper.

INPUT

ABC80 skriver frågetecken och väntar på inmatning till en talvariabel eller en sträng. Mellanslag och alla andra tecken med en lägre ascii-kod tas bort. Alla andra tecken ekas på skärmen. Inmatningen avslutas med RETURN. RETURN ger ny rad på skärmen. Koden för RETURN placeras inte i strängen.

INPUTLINE

Inget frågetecken skrivs ut. ABC80 väntar på inmatning avslutad med RETURN med en sträng som mottagare. Tecknen ekas (återskrivs) på skärmen. Mellanslag tas med, men inte tecken med

lägre ascii-kod. Markören på skärmen stannar efter sist ekade tecken. (Ej ny rad.)
Koderna för RETURN (13) och LINEFEED (10) placeras sist i strängen. Vill vi ta bort dessa kontrolltecken gör vi så här:

```
100 A$=LEFT$(A$,LEN(A$)-2)
```

GET

Inget frågetecken skrivs ut. Blinkande markör visar att ABC80 väntar på inmatning av ett enda tecken. Alla tecken (utom CTRL-C) kan ges in. Tecknen ekas ej.

Vid dessa tre instruktioner ligger programmet och väntar på tangenttryckning. Ascii-koderna (0 - 127) från tangentbordet ryms i 7 bitar. Så snart en tangent hålls nedtryckt ettställs den mest signifikanta biten, den med värde 128. Detta leder till att en avbrottsrutin utförs som hämtar in det aktuella tecknet.

5.2.3 POLLNING MED INP

Vi kan även kontinuerligt känna av tangentbordet. Tangentbordets adress är 56.

Exempel.

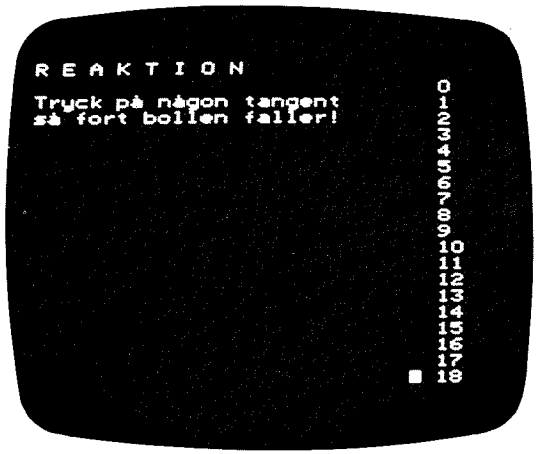
```
10 PRINT INP(56); : GOTO 10
RUN
```

En hel massa siffror dyker upp på skärmen. Det är ascii-koden för den sist nedtryckta tangenten. Lägg märke till att så länge en tangent hålls nedtryckt är värdet ascii-kod + 128. (Bit-7 ettställd).

Att polla tangentbordet med INP-instruktionen kan vara användbart när programmet ska göra någonting så länge som en tangent hålls nedtryckt eller till dess en tangent trycks osv.

Exempel från underhållningssektorn:

```
100 REM * Reaktion-ite * (Ctrl-C)
110 REM * Omgång-pos * (ej FUSK)
120 PRINT CHR$(12)"R E A K T I O N"
130 ; : ; "Tryck på någon tangent"
140 ; "så fort bollen faller!"
150 X=0 : REM Bollens koordinat
160 ; CUR(X,27)"¦"
170 REM * Vänta-ite * (RND-tid)
180 FOR I=0 TO 1000+2000*RND
190 IF INP(56)>=128 THEN 325 : REM Q(Omgång)
230 NEXT I
240 REM * Vänta-end *
250 ; CHR$(7); : REM TUT !
260 REM * Fall-ite * (Golv or Tangent)
270 IF X>23 OR INP(56)>=128 THEN 310
280 ; CUR(X,27)" "CUR(X+1,27)"¦"X;
290 X=X+1
300 GOTO 260
310 REM * Fall-end *
320 GOTO 330
325 REM * Omgång-adm * (FUSK)
326 REM - Tangent tryckt för tidigt
327 ; CUR(10,0)"FUSK !"
330 REM * Omgång-end *
335 FOR I=0 TO 5000 : NEXT I
340 GOTO 100
350 REM * Reaktion-end *
360 END
```



Exempel 5.1 Programmet REAKTION.

5.3 Bildskärmen

5.3.1 BILDMINNET

När vi skriver någonting på skärmen med PRINT, går det till så att det vi skriver lagras i en speciell minnesarea - bildminnet. Det speciella med den delen av ABC80´s minne är att det finns en särskild elektronikenhet - videogeneratorn - som kontinuerligt läser vad som står i bildminnet och omvandlar den informationen till videosignaler som går ut till monitorn. En direkt kopia av skärmen finns alltså i minnet, och om vi ändrar något där syns det direkt på skärmen.

Vi kan alltså skriva på skärmen genom att använda POKE. I appendix-D finns en minneskarta där vi ser att bildminnet upptar adresserna 31744 till 32767. På grund av konstruktionen hos videogeneratorn ligger dock inte skärmraderna konsekutivt i bildminnet, utan vi måste göra en avbildning från X och Y koordinater till en adress i minnet. Med hjälp av tabellen i appendix-E är det lätt att göra denna avbildning.

```
100 DIM R(23) : REM Tabell med radernas början
110 FOR I=0 TO 23 : READ R(I) : NEXT I
120 DATA 31744,31872,32000,32128,32256,32484,32512,32640
130 DATA 31784,31912,32040,32168,32296,32424,32552,32680
140 DATA 31824,31952,32080,32208,32336,32464,32592,32720
150 REM Nu kan vi omvandla koordinaterna
160 REM X och Y till adresser i bildminnet genom :
170 REM adress = R(X) + Y
180 REM Nedan följer två ekvivalenta rader
190 PRINT CUR(10,10)"*"
200 POKE R(10)+10,ASC("*")
```

Exempel 5.2 Adressering i bildminnet.

Den verkliga nyttan av denna möjlighet framgår av nästa avsnitt.

5.3.2 MARKÖREN - BLINKNING

Vi kan få vilket tecken som helst att blinka genom att ettställa den mest signifikanta biten (bit 7) i rätt cell i bildminnet.

Exempel:

```
10 ;CUR(0,0)"HEJ"
20 FOR I=31744 TO 31746
30 POKE I, PEEK(I) OR 128 : REM Sätt bit 7 till "1"
40 NEXT I
RUN
```

För att avsluta blinkningen behöver vi bara nollställa biten igen (POKE I, PEEK(I) AND 127).

Med hjälp av tabellen i förra avsnittet går det enkelt att skriva blinkande text på önskad plats på skärmen.

Observera att kontrolltecken (ASCII-kod < 32) inte blinkar även om vi ettställer blinkbiten. Om vi t ex fyller hela bildminnet med NUL-tecken (ASCII-kod 0) kommer även den blinkande markören att försvinna (tills skärmen rullar).

```
10 FOR I=31744 TO 32767
20 POKE I,0 : NEXT I
30 ;CUR(0,0);
```

5.3.3 GRAFIKEN

Bildskärmen har också en grafisk mod. I denna har vi en upplösning av 72 * 78 punkter, som vi kan tända eller släcka.

Kort sammanfattning:

PRINT CHR$(151);	Sätter resten av raden i grafisk mod.
PRINT CHR$(135);	Sätter tillbaka resten av raden till text mod.
SETDOT X,Y	Tänder den grafiska punkten (X,Y), där X kan vara 0 - 71 och Y kan vara 2 - 79.
CLRDOT X,Y	Släcker grafisk punkt i (X,Y).
DOT(X,Y)	Funktion som är sann (= -1) om den grafiska punkten i (X,Y) är tänd, och falsk (= 0) annars.

Punkten (0,0) ligger i övre vänstra hörnet, så koordinatsystemet ser ut så här:

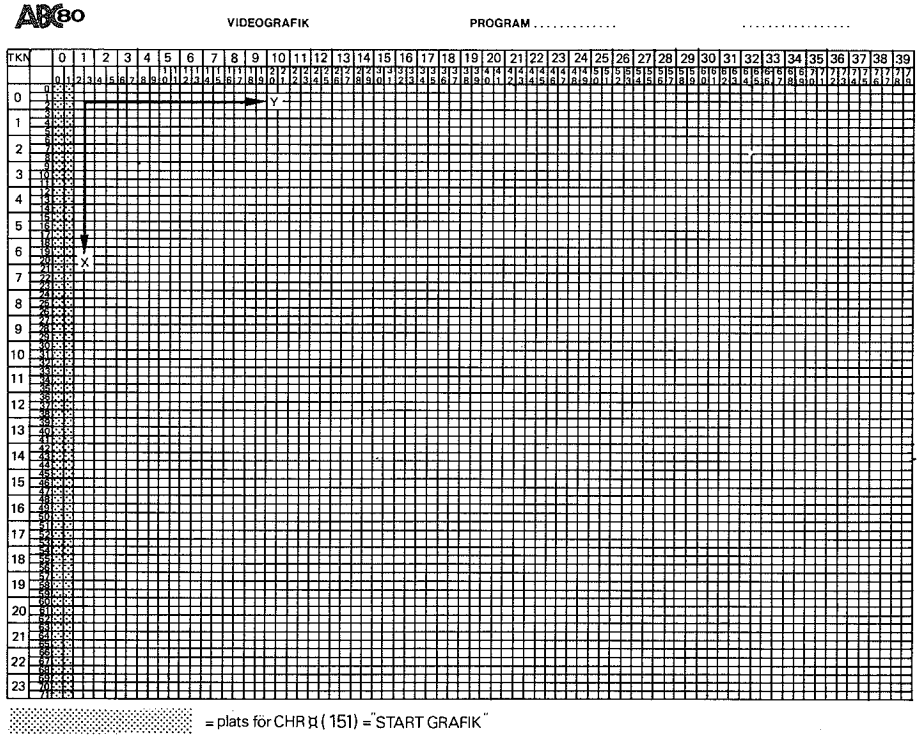


Fig 5.3 Skärmens axlar.

Observera att DOT(X,Y) kan vara SANN även där vi har skrivit text på skärmen!

Detta beror på att DOT(X,Y) testar om den bit i minnet, som koordinaterna (X,Y) pekar ut är ettställd, och det kan den ju vara utan att vi har en tänd grafisk punkt där (t ex om skärmen inte ens är satt i grafisk mod!).

5.3.4 ATT RITA BILDER

Vi kan rita grafiskt med SETDOT X,Y eller genom att kombinera lämpliga tecken i strängar och skriva ut dem.

Exempel.

```
10 ;CHR$(12,151);
20 FOR Y=2 TO 11
30 SETDOT 0,Y
40 NEXT Y
RUN
```

Detta lilla program ritar ett streck högst upp till vänster på skärmen. Vi kan göra samma sak med

```
NEW
10 ;CHR$(12,151)"#####"
RUN
```

Skillnaden här är att om vi i stället vill ha strecket från (1,3) till (1,12) behöver vi i det första fallet (SETDOT) bara ändra rad 20 och 30 :

```
20 FOR Y=3 TO 12
30 SETDOT 1,Y
```

Men i fallet med strängar måste vi titta i tabellen (appendix-E) och välja ut de tecken som ger oss rätt bild. Det blir i det här fallet strängen "(,,,,$". Dvs.

```
10 ;CHR$(12,151)"(,,,,$"
```

Slutsatsen blir att om vi vill rita en bild var som helst på skärmen är det enklast att använda SETDOT och ha en tabell med de koordinater som ska ritas. Om vi däremot har en bestämd bild, som alltid ska ritas på ett bestämt ställe på skärmen, kan det löna sig att använda strängar.

5.3.5 LAGRING AV BILDER

Det finns flera sätt att representera grafiska data (t ex vid lagring på fil). Beroende på vilken metod vi använder får vi skriva olika subrutiner som kan rita vår bild utgående från dessa data.

1. Vi sparar X och Y koordinater för varje punkt i bilden. Detta ger snabbt stora datamängder, men ´subrutinen´ blir enkel - SETDOT X,Y . För att kunna rita bilden var som helst på skärmen lagrar vi data för en bild högst upp i vänstra

hörnet, och kan sedan rita den var som helst genom att addera konstanter till X och Y värdena.

```
100 REM skärmen förutsätts vara satt
110 REM i grafisk mod.
120 READ A : REM antal punkter
130 FOR I=1 TO A
140 READ X,Y
150 GOSUB 290 : REM plotta punkten
160 NEXT I
170 GOTO 170
180 DATA 12
190 DATA 3,3,3,4,3,5,3,6
200 DATA 4,3,4,6,5,3,5,6
210 DATA 6,3,6,4,6,5,6,6
220 REM
230 REM ------------------------------
240 REM - Subrutin för att rita en punkt
250 REM
260 REM Testa först om koordinaterna
270 REM är på skärmen.
280 REM
290 IF (X<0) OR (X>71) OR (Y<2) OR (Y>79) THEN 370
300 REM
310 REM --- Tänd punkten
320 REM
330 SETDOT X,Y
340 REM
350 REM --- Klart
360 REM
370 RETURN
```

Exempel 5.3 Punktgenerator.

2. Vi anger bara start och slutpunkter för räta linjer och låter programvaran generera linjerna. I en figur som består mest av räta linjer ger detta en betydande inbesparing av data, men om bilden består mest av enstaka punkter går det åt mer än i metod 1 (även en punkt måste ju specificeras med fyra värden)

```
100 ; CHR$(12)
110 FOR I=0 TO 23 : ; : ; CHR$(151); : NEXT I
120 READ A : REM antal linjer
130 FOR I=1 TO A
140 READ X1,Y1,X2,Y2
150 GOSUB 300
160 NEXT I
170 GOTO 170
175 DATA 4
180 DATA 2,2,2,6
190 DATA 3,6,5,6
200 DATA 5,5,5,2
210 DATA 4,2,3,2
220 REM
230 REM ------------------------------
240 REM - Subrutin för att rita linjer
250 REM
260 REM - (X1,Y1) och (X2,Y2) är
270 REM - start resp. slutpunkter.
280 REM
290 REM --- Rita första punkten
300 SETDOT X1,Y1
301 REM
302 REM --- Testa om bara en punkt
304 REM
306 IF (X1=X2) AND (Y1=Y2) THEN 600
308 REM
309 REM --- Rita
310 X3=X2-X1 : Y3=Y2-Y1
320 IF X3>0 THEN X4=1 ELSE X4=-1
330 IF Y3>0 THEN Y4=1 ELSE Y4=-1
340 X3=ABS(X3) : Y3=ABS(Y3)
345 REM
350 REM --- Testa om fall 1 eller 2
360 REM
370 IF X3>Y3 THEN 500
375 REM
380 REM --- Fall 1 :  Y3 större än X3
390 REM
400 S=X3/Y3
410 FOR S1%=1% TO Y3
420 R=R+S
430 IF R>=1 THEN X1=X1+X4 : R=R-1
440 Y1=Y1+Y4
450 SETDOT X1,Y1
460 NEXT S1%
470 GOTO 600
```



```
480 REM
490 REM --- Fall 2 :  X3 större än Y3
495 REM
500 S=Y3/X3
510 FOR S1%=1% TO X3
520 R=R+S
530 IF R>=1 THEN Y1=Y1+Y4 : R=R-1
540 X1=X1+X4
550 SETDOT X1,Y1
560 NEXT S1%
570 REM
580 REM --- Klart
590 REM
600 RETURN
```

Exempel 5.4 Linjegenerator.

3. Ett helt annat alternativ är att lagra strängar. Detta lämpar sig bäst för fasta bilder på ett bestämt ställe på skärmen.

```
100 ; CHR$(12)
110 FOR I=0 TO 23 : ; : ; CHR$(151); : NEXT I
120 READ A : REM antal strängar
130 FOR I=1 TO A
140 READ X$
150 GOSUB 270 : REM rita med strängen
160 NEXT I
170 GOTO 170
180 DATA 2
190 DATA ´!!7k´,´"!##´
200 REM
210 REM ------------------------------
220 REM - Subrutin för att rita med en sträng
230 REM
240 REM - De första två tecknen i strängen
250 REM - är koordinater (0-23, 0-39)
260 REM
270 REM --- Rita
280 REM
290 ; CHR$(27,61)+X$;
300 REM
310 REM --- Klart
320 REM
330 RETURN
```

Exempel 5.5 Stränglagrade bilder.

I alla tre fallen ovan kan data lika gärna finnas på en fil som vi läser från istället för DATA-satser. Alla READ byts då ut mot INPUT #.

Det mest kompakta sättet att lagra grafiska data är strängalternativet, men som nämnts ovan är det då lite svårt att flytta bilden.

Man kan också packa data på något sätt. T ex vet vi att X och Y koordinaterna aldrig är mindre än 0 eller större än 79. Det är ju slöseri att låta detta ta upp ett helt heltal (som ju kan vara -32768 till +32767). Vi kan använda oss av SWAP% för att enkelt lagra två koordinater i ett heltal.

Exempel på två små subrutiner för detta:

```
1010 REM --- Packning av X och Y till D%
1020 D%=SWAP%(X)+Y : RETURN



1030 REM --- Uppackning av D% till X och Y
1040 X=SWAP%(D%) AND 255%
1050 Y=D% AND 255% : RETURN
```

Med detta har vi reducerat utrymmesbehovet för våra koordinatdata till hälften, men vi måste i stället ha lite mera program (som dessutom tar lite längre tid att köra).

5.4 Ljudgeneratorn

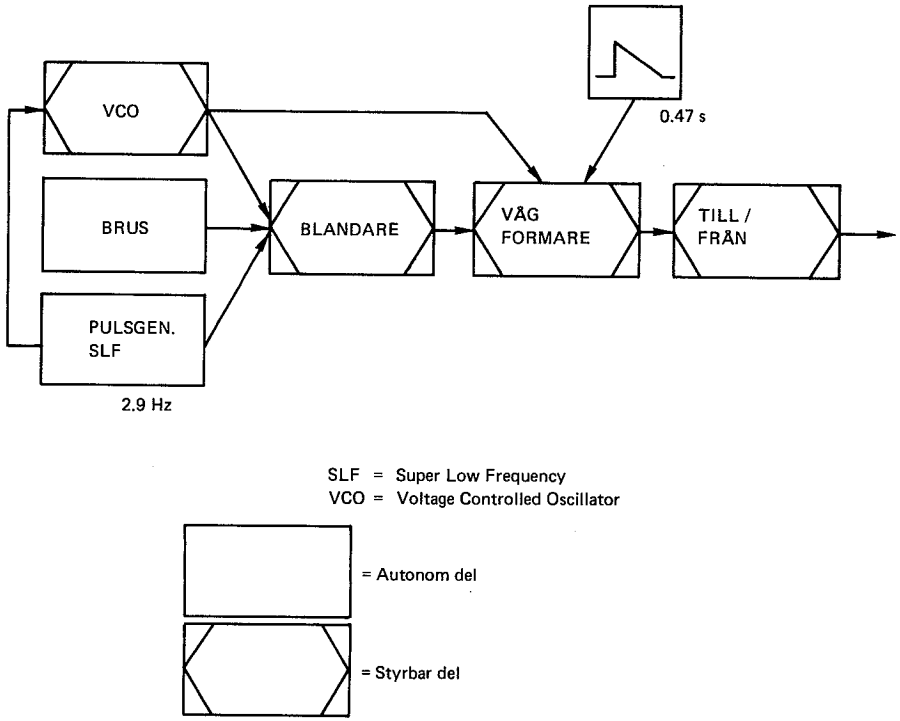


Fig 5.4 Ljudgeneratorn.

Denna bild återger schematiskt ljudgeneratorn i ABC80. Med ett "styrord" på portadress 6 kan vi forma olika ljud.

VÅGFORMARE		BLANDARE			VCO		TILL / FRÅN
b7	b6	b5	b4	b3	b2	b1	b0
128	64	32	16	8	8	2	1
0	0	Ingen funktion					
0	1	Ingen funktion					
1	0	Avklingande ljud					
1	1	VCO tas som vågformare					

0	0	0	VCO
0	0	1	BRUS
0	1	0	SLF
0	1	1	VCO + BRUS
1	0	0	BRUS pulsat
1	0	1	VCO pulsad
1	1	0	VCO + BRUS pulsat
1	1	1	Från

Hög ton	0	0
Låg ton	0	1
SLF-styrd	1	0
SLF-styrd	1	1

Från	0
Till	1

Fig 5.5 Hur styrordet påverkar ljudgeneratorn.

Det är inte helt trivialt att åstadkomma ett visst önskat ljud. Av alla 256 kombinationer som styrordet kan anta resulterar ca 34 i olika ljud. Det kan vara bekvämt att göra upp en tabell med dessa ljud:

Styrord	Beskrivning
	-RENA KOMPONENTER
1	Ren ton, högsta frekvens
249	.
3	.
251	Ren ton, lägsta frekvens

5	Hög frekvens, siren ´oioioi´
253	Låg frekvens, siren ´oioioi´
9	Brus ´Vattenfall´
17	´knäpp-knäpp-knäpp´ (Fiskebåt)
	-RENA KOMPONENTER + BRUS
25	Högsta frekvens
217	.
27	.
219	Lägsta frekvens
29	Hög frekvens, siren ´oioioi´
221	Låg frekvens siren ´oioioi´
	-PULSAT MED SLF
41	Högsta frekvens
209	.
43	.
211	Lägsta frekvens
45	Hög frekvens, pulsad siren ´piop-piop-piop´
237	Låg frekvens, pulsad siren
33	Brus, pulsat (Tuff-tuff-lok)
	-PULSAT MED SLF + BRUS
49	Högsta frekvens
241	.
51	.
243	Lägsta frekvens
55	Hög frekvens, pulsad siren ´piop-piop-piop´
247	Låg frekvens, pulsad siren
	-KORT LJUDPULS
129	Hög frekvens
131	Låg frekvens (=Bell, ;CHR$(7))
135	´oi...´ (Olika varje gång)
137	Brus (Ett skott)
	-KORT LJUDPULS MED BRUS
153	Hög frekvens
155	Låg frekvens
157	´oi...´ (Olika varje gång)

Fig 5.6 Användbara ljud.

Genom att ge korta ljudstötar eller genom att snabbt växla mellan två - tre olika ljud kan vi få fram andra effekter.

```
10 ; CHR$(12)
20 ; CUR(10,8)´Veckans pausfågel´
30 ; CUR(12,8)´Trädgårdsångaren´
40 FOR K=100*RND TO 100*RND
50 OUT 6,5
60 FOR I=0 TO K : NEXT I
70 OUT 6,0
80 FOR I=0 TO K : NEXT I
90 NEXT K
100 GOTO 40
```

Exempel 5.6 "Veckans pausfågel".

Skämt åsido utgör faktiskt ljudet en viktig del av dialogen människa - dator. Speciellt när det gäller att påkalla uppmärksamhet, vid felmeddelande eller larm.

5.5 Dialogprogram

5.5.1 RULLANDE SKÄRM

På en bildskärm kan vi skriva på samma sätt som om vi hade en skrivande terminal med pappersrulle. Då programmet skriver utan att positionera markören och t ex begär inmatning av ett fält per rad rullar bilden uppåt (eng. scroll). Denna metod ger oss enkla och små program.

Jämfört med en skrivande terminal finns dock en avgörande skillnad: Den översta raden försvinner från skärmen. Är det många uppgifter som ska matas in förlorar vi lätt överblicken, speciellt om vi svarar fel några gånger rullar de tidigare raderna snart bort.

5.5.2 MENYTEKNIK FÖR VAL AV RUTIN

Små program utför måhända bara en enda uppgift. I större program eller programsystem är det lämpligt att dela upp programmet i ett

antal fristående rutiner. Vi presenterar en meny (lista) över de rutiner som ingår och låter operatören välja. Ett sätt är att ge varje rutin ett nummer som operatören får välja med tangenterna 1 till 9. Andra möjligheter beskrivs i ref-7.

Menyer kan vi ha på flera nivåer.

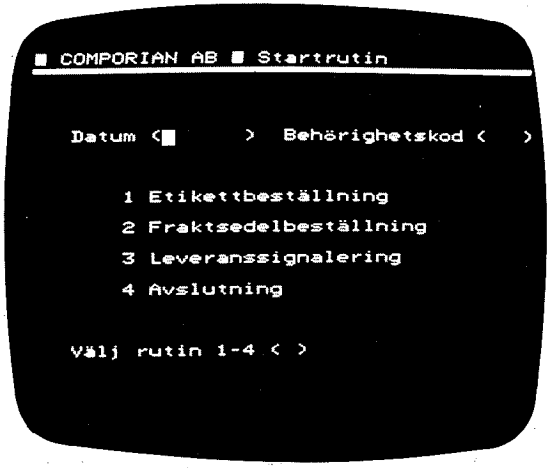


Fig 5.7 Exempel på meny.

```
10 Q1$=´COMPORIAN AB´
20 Q2$=´Startrutin´
30 DATA 4
40 DATA ´Etikettbeställning´
50 DATA ´Fraktsedelbeställning´
60 DATA ´Leveranssignalering´
70 DATA ´Avslutning´
80 RESTORE 30
90 GOSUB 5320 : REM *Meny*
100 ON GO% GOSUB 1000,2000,3000,4000
110 GOTO 10
.
.
.
5000 REM ---
5010 REM ¦ Systemrad
5020 REM ---
5030 REM *Sysrad*
```

```
5040 ; CHR$(12%)´ ¦ ´Q1$´ ¦ ´Q2$
5050 ; CHR$(151%)´#####################################´
5060 REM *Sysrad-end*
5070 RETURN
5080 ;
5090 REM ---
5100 REM ¦ Meddelanderad
5110 REM ---
5120 REM *Medd*
5130 ; CUR(23%,2%)TAB(39%)CUR(23%,2%);
5140 REM *Medd-end*
5150 RETURN
5160 ;
5170 REM ---
5180 REM ¦ Menyval
5190 REM ---
5200 REM Visa meny och välj
5210 REM Invariabler och datasatser :
5220 REM - GO$  Rutin-namn
5230 REM - DATA <ant. rutiner>
5240 REM - DATA <Rutinnamn-1>
5250 REM - DATA <Rutinnamn-2>
5260 REM - ...
5270 REM Utvariabler :
5280 REM - G1   Antal rutiner
5290 REM - GO   Valt rutin-nr.
5300 REM - GO$  Nytt rutin-namn
5310 REM ---
5320 REM * Meny-seq *
5330 GOSUB 5030 : REM * Sysrad *
5340 ; CUR(2%,1%)GO$
5350 READ G1%
5360 FOR I%=1% TO G1%
5370 READ FO$(I%) : ; CUR(2%+2%*I%,3%)I%´ ´FO$(I%)
5380 NEXT I%
5390 GOSUB 5120 : REM * Medd-rad *
5400 ; ´Välj rutin (1 -´G1%´) ´;
5410 GET A$ : A%=ASC(A$)
5420 GO%=A%-48%
5430 IF GO%<1% OR GO%>G1% THEN 5390
5440 GO$=FO$(GO%) : ; CUR(2%+2%*GO%,0%)´->´
5450 GOSUB 5140 : REM *Medd*
5460 REM * Meny-end *
5470 RETURN
```

Exempel 5.7 Subrutin för att visa meny och välja rutin.

5.5.3 FORMULÄRTEKNIK

Istället för rullande skärm kan vi använda oss av formulärteknik. Ett formulär består av ett antal ledtexter som visas på skärmen och som operatören får fylla i. Alla fält får sin givna plats på skärmen vilket ger en bra översikt. Se figur 5.2 .

Principen är alltså denna:
Vi utgår från en meny av rutiner. Till varje rutin hör ett eller flera formulär där vi gör den egentliga inmatningen av fält eller ger kommandon. När vi genomfört en rutin bör vi komma tillbaka till meny-bilden.

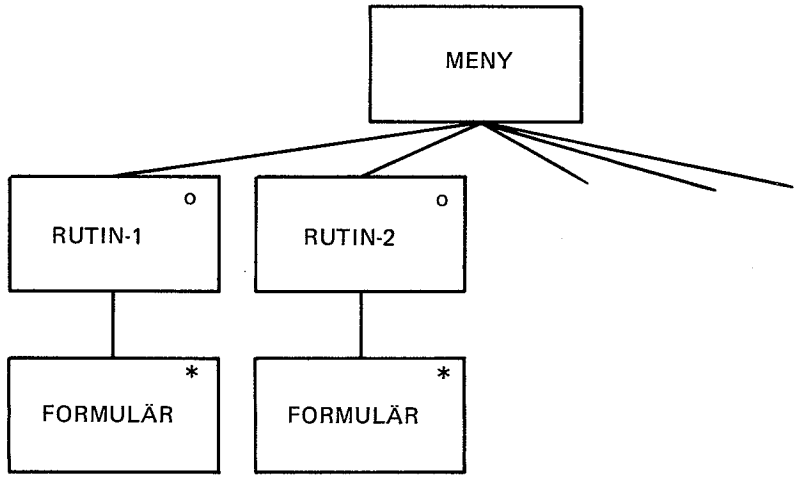


Fig 5.8

Jämför med en bok där vi först slår upp innehållsförteckningen och därefter väljer det kapitel som är av intresse. Kapitlet kan bestå av en eller flera sidor.

Formulärteknik ger, rätt utnyttjad, en trevlig användarvänlig dialog. Programmet för att åstadkomma detta blir dock väsentligt mera omfattande än vid rullande skärm.

För att peka ut det fält som står i tur att ges in måste vi placera markören efter motsvarande ledtext. Vi behöver alltså en tabell över var varje fält börjar på skärmen. Vidare måste vi kontrollera fältlängderna så att t.ex inte för många tecken ges in och

formuläret skrivs sönder. Vi kan behöva backa tillbaka till ett tidigare fält och behöver en tangent för detta. Vissa fält vill vi gå förbi. Vissa fält får vi inte gå förbi då de är obligatoriska. Vi kanske vill kunna "bläddra" oss fram till nästa formulär, - eller vi valde fel rutin och vill "bläddra" oss tillbaka till menyn utan att mata in någonting.
Det hela är faktiskt inte så svårt som ovanstående kanske ger intryck av. Lösningen består i att man för varje fält specificerar ett antal parametrar som bestämmer markörsposition, min. och max. fältlängd, vilka funktionstangenter som får tryckas etc.

Dessa parametrar kan, liksom formulärens ledtexter, lagras som DATA-satser i programmet. Vid stora program med många rutiner och, formulär och fält kan parametrar och formulärstexter istället lagras på en diskett-fil. Därigenom kan vi få in mycket omfattande program i ABC80.

5.5.4 DISPONERING AV BILDSKÄRMEN

För att användaren lätt ska känna igen sig på skärmen bör vi ha vissa regler om VAD som ska finnas VAR på den.

Systemrad

Vissa uppgifter, t ex vilket program som vi arbetar med, bör visas på alla skärmbilder. Andra specifika uppgifter kan vara dagens datum, vilken data-diskett som programmet arbetar med etc. Dessa uppgifter sammanför vi till en systemrad som vi alltid visar på rad 0.

Rutinrad

Vi bör även veta vilken rutin vi befinner oss i efter att ha gjort ett meny-val. Den rad vi valde ur menyn visar vi på rad 2 eller möjligen till höger på systemraden om det finns plats.

Rad 3 till 22 disponerar vi fritt för formulär m.m.

Meddelanderad

Rad 23, den nedersta raden, avsätter vi som meddelanderad för felmeddelanden och andra informationer.

5.5.5 FELMEDDELANDEN

För att inte störa dialogen med onödiga programavbrott bör vi söka ta hand om BASIC-felen med instruktionen ONERRORGOTO på det sätt som vi gått igenom i kapitel 2. Vi sätter då variabeln E9=ERRCODE.

Vi behöver dessutom ett antal felmeddelanden som är knutna till det program som vi utarbetar, dvs. som är speciella för tillämpningen. Vi numrerar lämpligen dessa från 100 och uppåt för att inte förväxla dem med ERRCODE. E9=100, 101, 102 osv. är alltså fel i vår applikation. Motsvarande meddelandetexter skriver vi som DATA-satser.

Vi bör ha felmeddelanden i klartext!

På nästa sida kan vi hitta ett exempel på en felrutin som visar båda typer av felmeddelanden. Längst till vänster på meddelanderaden visas ett blinkande ">" vid fel. Ett "pip" i högtalaren ges också.

Rutinen går att använda såväl vid tekniken med rullande skärm som vid formulärteknik. Vid formulärteknik bör markören, efter att felmeddelandet har presenterats, peka ut det fält där felet uppstod.

```
100 DATA "Programfel"
101 DATA "Fel tangent"
102 DATA "För kort fält"
103 DATA "För långt fält"
104 DATA "Siffror !"
105 DATA "Bokstäver!"
106 DATA "Svara J/N!"
107 DATA "Får ej rättas"
108 DATA "För litet tal"
109 DATA "För stort tal"
110 DATA "Felaktigt tal"
.
.
.
6000 REM ---
6010 REM ¦ Fel-meddelande
6020 REM ---
6030 REM Visar felmeddelande
6040 REM Invariabel :
6050 REM - E9  Felnummer eller 0
6060 REM ---
6070 REM * Fel-sel * (E9<>0)
6080 IF E9%=0% THEN 6210
6090 ; CUR(23%,2%)TAB(39%)CUR(23%,2%);
6100 POKE 32720%,ASC(">")+128% : REM Blink
6110 ; CHR$(7%); : REM Pip
6120 REM * Feltyp-sel * (E9>=100)
6130 IF E9%<100% THEN 6180
6140 RESTORE 100
6150 FOR I%=100% TO E9% : READ B$ : NEXT I%
6160 ; " "B$;
6170 GOTO 6200
6180 REM * Feltyp-or *
6190 ; "Fel nr"E9%" (Se fellista)";
6200 REM * Feltyp-end *
6210 REM * Fel-or *
6220 REM * Fel-end *
6230 RETURN
```

Exempel 5.8 Subrutin för felmeddelanden.

KAPITEL 6

Programmet procent -Ett större exempel

ABC80

6.1 Problemet

Nu har vi gått igenom så mycket att det är dags att använda våra kunskaper praktiskt. Det gäller såväl programkonstruktion, filer som interaktion människa - dator.

Många program arbetar enbart med heltal. Det kan dock vara rätt arbetsamt att sätta %-tecken på alla variabler och konstanter när vi skriver ett BASIC-program.

Låt oss göra ett program som fixar det automatiskt! Programmet PROCENT.

Hur går man tillväga?

Vi utgår ifrån att det program som vi ska bearbeta med PROCENT finns lagrat på diskett. Det bör vara lagrat med LIST som en textfil (.BAS). Denna fil blir indata till programmet PROCENT som sedan ska skapa en ny fil där alla flyttalsvariabler och konstanter fått %-tecken påhängt.

Observera att infil och utfil visserligen är program (som går att ladda och köra), men ur PROCENT´s synpunkt är de data.

PROCENT måste alltså kunna känna igen variabelnamn bland allt annat som finns i ett program: IF ... THEN ... ON ... DATA ..... - och konstanter.

Värre blir det med konstanter!
Det finns ju en massa radnummer också i ett program, dem får vi inte sätta %-tecken på.

Hmm! Det kanske inte blir så lätt!

Det här är nog ett problem där man kan trassla till det ganska ordentligt för sig om man inte strukturerar det.

Låt oss använda JSP.

6.2 De fyra stegen

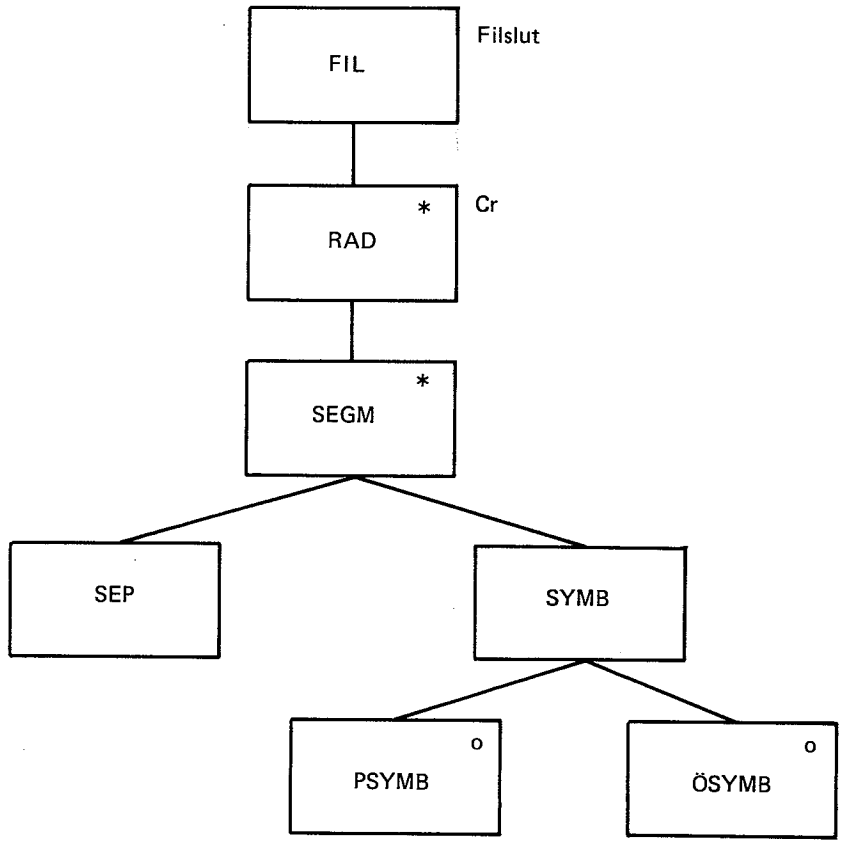


Fig 6.1 Datastruktur infil.

6.2.1 STEG 1: RITA DATASTRUKTUR

Vi börjar med att rita en första datastruktur över infilen som PROCENT ska behandla.

Vår infil är en iteration av rader. En rad består av någonting vi kallar symboler med någon form av separatorer emellan. Varje par

(sekvens) separator-symbol kallar vi segment. (Det svåraste med JSP är att hitta på namn till alla rutor.) En rad avslutas med koden för RETURN, CR.

Symboler slutligen är antingen sådana som ska ha %-tecken (PSYMB) eller övriga (ÖSYMB).

Stränguttryck är otrevliga! I dessa kan det stå vad som helst. Låt oss därför klassa allt som står mellan strängtecken (" eller ´) som separatorer.

En rad börjar alltid med radnummer så låt oss först rita in detta. Se datastruktur i appendix-F.

REM- och DATA-satser måste vi hoppa över. De kan stå ensamma på en rad eller i slutet på en rad. Utmärkande för dem är att de alltid sträcker sig till radens slut (CR).

Lämpligen beskriver vi en rad som en sekvens av RADNR, RADBÖRJAN, RADSLUT och CR.

RADBÖRJAN är en iteration av delar av raden som ska bearbetas. RADSLUT är antingen REM-/DATA-sats eller ingenting alls.

Följ med i appendix-F!

Det problem som återstår är att skilja ifrån de radnummer som finns inuti satser. Var finns dessa?
Jo efter GOTO, GOSUB, RESTORE och ONERRORGOTO, - och så THEN och ELSE förstås. Men efter THEN och ELSE bara ibland.

Hmm! Bäst att titta på THEN/ELSE för sig. Vi låter BEARB vara en selektion mellan det som börjar med symbolerna THEN eller ELSE och resten (BSYMB).

Efter THEN eller ELSE kommer antingen RADNR eller något vi måste titta närmare på (BSYMB). Som vi ser förekommer BSYMB på två ställen.

I BSYMB skiljer vi först på de symboler som har radnummer efter sig. Det kan eventuellt vara flera radnummer, som t ex vid ON .. GOTO, varför vi får en iteration av radnummer fram till satsens slut.

Det vi nu har kvar att välja ut är symboler som består av en bokstav (A), en bokstav och en siffra (A9) eller enbart av

siffror (9). Dvs flyttalsvariabler eller flyttalskonstanter. Är så fallet ska dessa symboler ha %-tecken, annars inte (Övr)!

Datastrukturen för in-/ut-fil är inte den enda som programmet PROCENT måste bygga på. Programmet ska ju även kommunicera med operatören via tangentbordet och bildskärmen. Låt oss rita en lämplig datastruktur även för detta. Appendix-F, punkt 2.2.

Vi börjar med att visa en förklarande text på skärmen.
Kanske kan det vara lämpligt att vi kan bearbeta flera programfiler utan att behöva gå ur programmet. Vi gör därför en iteration av JOB. JOB avslutas med frågan "Vill du fortsätta?".

Namnet på in- och ut-filen måste ges in från tangentbordet. Som en indikation på att programmet löper och på hur långt det har kommit visar vi på skärmen numret på den rad som för tillfället bearbetas. Jobbet (JOB) avslutas, om allt gick väl med "Jobbet klart", annars med "Jobbet aborterat".

6.2.2 STEG2: FÖRENA DATASTRUKTURER TILL PROGRAMSTRUKTURER

Vi ritar först en grovstruktur för programmet baserad på de datastrukturer vi fick fram i steg 1. Se figur 6.2 .

Exekveringen kan gå fel av någon anledning, t ex läsfel på skivan eller då skivan blir full. Programmet bör ta hand om sådana fel.

Då vi behöver göra en selektion utan att ha underlag för att göra valet, använder vi en s.k. posit/admit-struktur. Se BHFIL i figur 6.2 . När vi börjar behandla filen vet vi ju inte om det ska gå vägen eller inte. Vi antar att det ska gå bra och börjar i den vänstra rutan (posit). Om det visar sig att vi fick fel får vi erkänna detta och gå över till den högra rutan (admit).

Ett sådant hopp, eller quit som det heter, är tillåtet i JSP. En quit sker alltid till motsvarande admit-del på samma nivå.

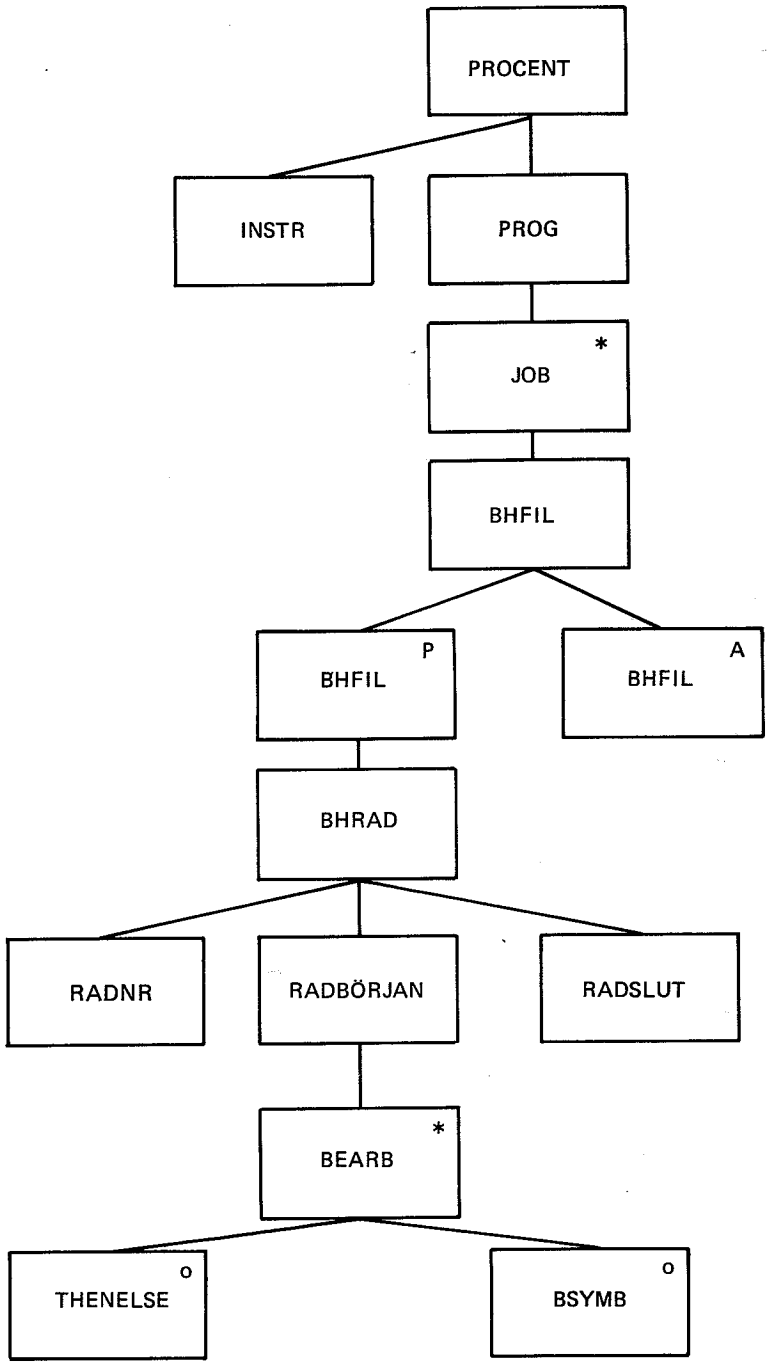


Fig 6.2 Grov programstruktur.

6.2.3 STEG 3: LISTA OPERATIONER OCH PLACERA I PROGRAMSTRUKTUREN

Vi är nu mogna för att rita den fullständiga programstrukturen och placera ut operationerna i denna. Först bör vi dock komplettera databeskrivningen med en beskrivning av de dedicerade variabler vi tänker använda. Se appendix-F, punkt 2.3 .

Vi använder oss av s.k. read-ahead-teknik som finns beskriven i Ref-8. Termineringsvillkor, selekteringsvillkor, eventuella quits och operationerna listar vi i anslutning till programstrukturen.

De generella operationerna beskriver vi för sig, punkt 4 i appendix-F.

6.2.4 STEG 4: KODA

Hur man kodar en JSP-struktur har vi lärt oss i kapitel 2. Det kan vara lämpligt att stycka upp huvudprogrammet i mindre delar m h a subrutiner för att få en bättre överblick. Av BHFIL gör vi därför en subrutin, trots att den bara förekommer en gång. Se programlistan i appendix-F.
Vi skulle kunna ha gjort samma sak med BEARB.

Operationerna kodar vi i fritt format. Vi bör dock bemöda oss om att även göra operationerna lättlästa. Lägg särskilt märke till den generella operationen NYSYMB där vi kostar på oss en JSP-struktur. Vi började givetvis med att rita en datastruktur för uppdelningen separator - symbol!

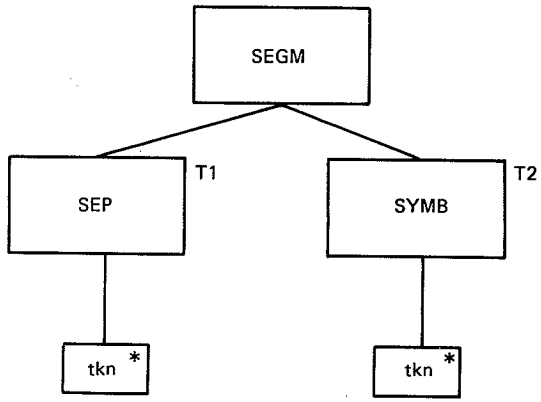


Fig 6.3

T1 CR or ´:´ or BOKSTAV or SIFFRA T2 not BOKSTAV and not SIFFRA and not ´$´ and not ´:´ and not ´%´

6.3 Vi testar

Det allra trevligaste med JSP-strukturerade program är att programmen oftast fungerar utan att man måste testa särskilt mycket. Eftersom vi tvingades tänka igenom programmet redan i konstruktionsfasen tog vi hand om de flesta problem redan då. Där är det lättare att ta hand om dem. Vi kan prova vårt program på sig självt (!) genom att ange PROCENT.BAS som infil. PROCENT innehåller emellertid inte så många flyttalsvariabler, men det blir ändå ca 25% snabbare i heltalsvarianten.

Vi provar också med programmet REAKTION som vi gjorde i förgående kapitel. Det blir mer än dubbelt så snabbt och tar nästan 100 bytes mindre plats !

Provkör! - Du har inte en chans att hänga med.

```
100 REM * Reaktion-ite * (Ctrl-C)
110 REM * Omgång-pos * (ej FUSK)
120 PRINT CHR$(12%)"R E A K T I O N"
130 ; : ; "Tryck på någon tangent"
140 ; "så fort bollen faller!"
150 X%=0% : REM Bollens koordinat
160 ; CUR(X%,27%)"¦"
170 REM * Vänta-ite * (RND-tid)
180 FOR I%=0% TO 1000%+2000%*RND
190 IF INP(56%)>=128% THEN 325 : REM Q(Omgång)
230 NEXT I%
240 REM * Vänta-end *
250 ; CHR$(7%); : REM TUT !
260 REM * Fall-ite * (Golv or Tangent)
270 IF X%>23% OR INP(56%)>=128% THEN 310
280 ; CUR(X%,27%)" "CUR(X%+1%,27%)"¦"X%;
290 X%=X%+1%
300 GOTO 260
310 REM * Fall-end *
320 GOTO 330
325 REM * Omgång-adm * (FUSK)
326 REM - Tangent tryckt för tidigt
327 ; CUR(10%,0%)"FUSK !"
330 REM * Omgång-end *
335 FOR I%=0% TO 5000% : NEXT I%
340 GOTO 100
350 REM * Reaktion-end *
360 END
```

Exempel 6.1 Programmet REAKTION i heltalsvariant.

KAPITEL 7

En A/D-omvandlare i BASIC

ABC80

7.1 Anpassning till omvärlden

Ett viktigt användningsområde för datorn är mätsystem. I ett sådant kan datorn t ex samla in mätvärden, göra statistik och varna för felaktiga värden, men den kan också styra själva mätförfarandet; ställa in mätområdet etc.

För detta behövs en anpassning (ett interface) mellan datorn och omvärlden - både elektriskt och programmässigt.

Eftersom datorn bara kan ta emot digitala data ("ettor" och "nollor"), måste också någon form av signalomvandling göras.

Tag som exempel att vi vill mäta spänningen på vårt gamla bilbatteri (under belastning). Den kan vara vad som helst mellan 0 V och 12 V. Denna spänning är analog och måste omvandlas till en form som datorn förstår.

Vi måste ansluta en Analog/Digital - omvandlare (A/D - omvandlare), dvs. en anordning som tar in batterispänningen i den ena änden, och ger ut ett (binärt) digitalt värde i den andra.

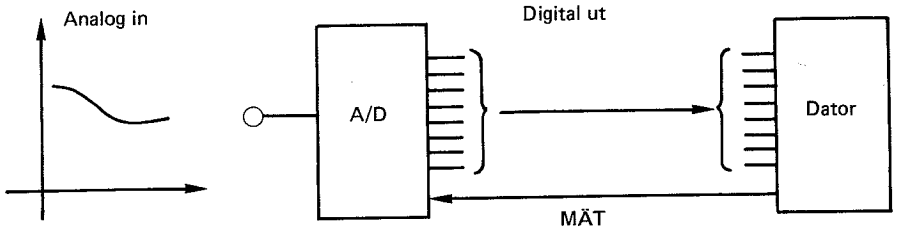


Fig 7.1

Om spänningen på batteriet är 11.7 V kan det på den digitala utgången stå t ex 117 (01110101 binärt).

En annan - enklare - metod för A/D - omvandling är att den digitala utgången ger en puls vars längd är proportionell mot den analoga inspänningen.

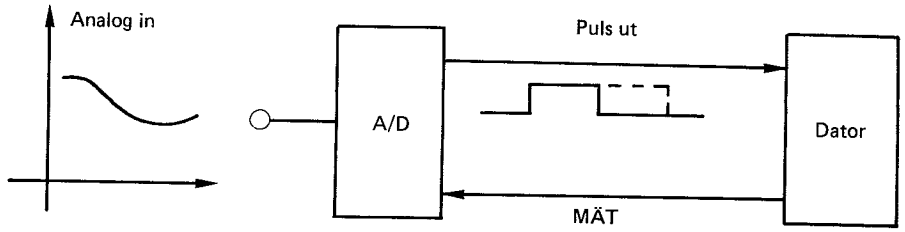


Fig 7.2

Med samma exempel som tidigare: 11.7 V in kan t ex motsvaras av en puls med längden 117 millisekunder.

Här behövs bara en tråd från A/D-n till datorn i stället för åtta, men i gengäld får datorn jobba lite mer - den måste mäta längden på pulsen.

Denna senare metod ska vi använda oss av i det här kapitlet.

7.2 Elektrisk anslutning av en JOYSTICK

Den enhet vi ska beskriva här är en JOYSTICK, dvs en sådan här manöverspak som finns till de flesta TV-spel. Den kan röras åt alla håll, och kan t ex användas för att rita bilder på skärmen.

Vi börjar med att titta på A/D - omvandlaren:

Det vi är intresserade av är alltså att överföra läget hos en spak till ett numeriskt värde (pulslängd).

Det finns ett digitalt byggelement som lämpar sig utmärkt för detta ändamål - den monostabila vippan, som har den trevliga egenskapen att när vi ändrar ingången från "nolla" till "etta" (triggar kretsen), så kommer vi på utgången att få en puls, vars längd enbart beror på den kondensator (C) och det motstånd (R) vi har anslutit till kretsen (se fig. 7.3).

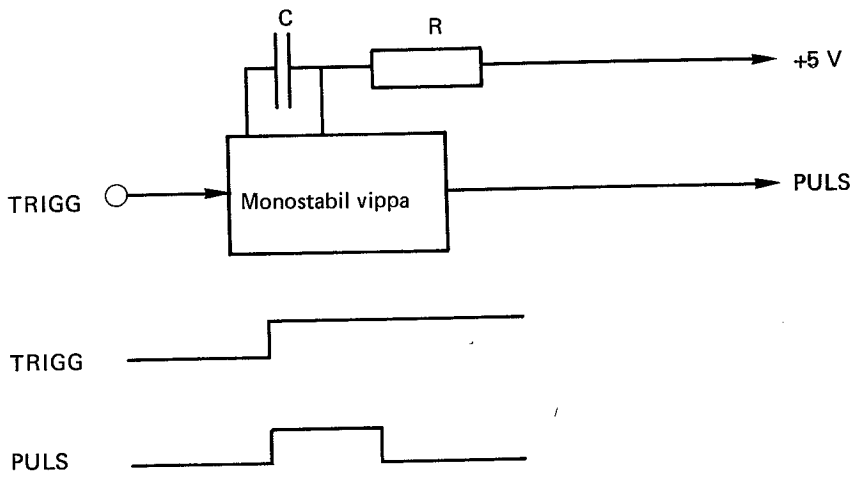


Fig 7.3

Närmare bestämt lyder förhållandet så:

PULSLÄNGD (sek.) = 0.45 * R (Ohm) * C (Farad)

Ett exempel:

R = 100 kiloohm
C = 10 mikrofarad

ger pulslängden = 0.45 sekunder

Men än så länge får vi alltid samma pulslängd, och det är ju inte mycket till A/D-omvandlare.

Vi byter således ut R mot ett variabelt motstånd, en potentiometer.

Om vi väljer en termistor (temperaturkänsligt motstånd) som det variabla motståndet så har vi fått en termometer! Det enda som behövs är lite kalibrering, så har vi ett instrument som kan mäta temperaturer (på begäran från datorn). Vi har fått ett mätsystem!

Genom att välja en lämplig kondensator, och lämplig resistans på det variabla motståndet, kan vi få pulslängden att variera inom de gränser vi vill ha den.

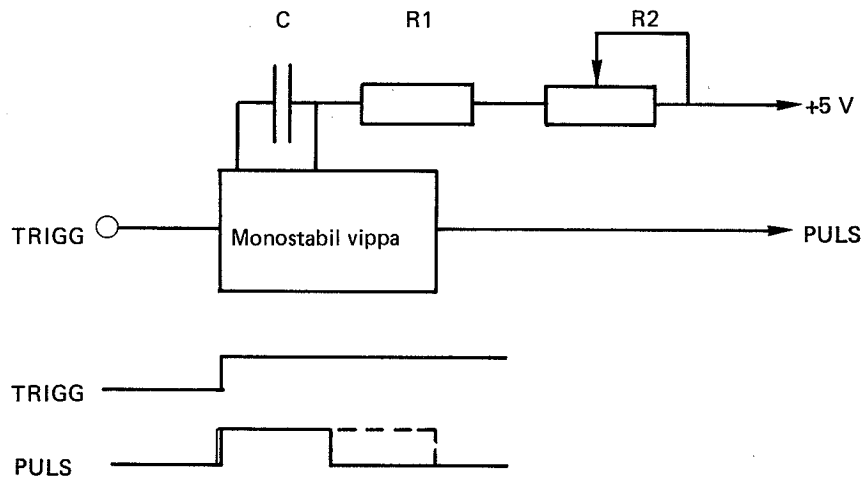


Fig 7.4

R1 finns med för att bestämma den kortaste pulslängden.

Lämpliga värden på komponenterna:

C = 6.8 mikrofarad
R1 = 4.7 kiloohm
R2 = 0-50 kiloohm
Monovippan : SN74LS123

Med dessa värden får vi pulslängder som kan variera mellan:

minst : 0.45 * 6.8 * 10E-6 * 4.7 * 10E3 = 0.014 sek.

mest : 0.45 * 6.8 * 10E-3 * 54.7 * 10E3 = 0.17 sek.

(Dessa pulslängder är valda för att passa BASIC-programmet i avsnitt 7.3)

Hjärtat i JOYSTICKen är två sådana A/D - omvandlare (en för X-koordinaten, en för Y-koordinaten), och två potentiometrar som styrs av spaken, den ena potentiometern vrids när vi rör spaken i sidled, och den andra när vi rör spaken fram och tillbaka.

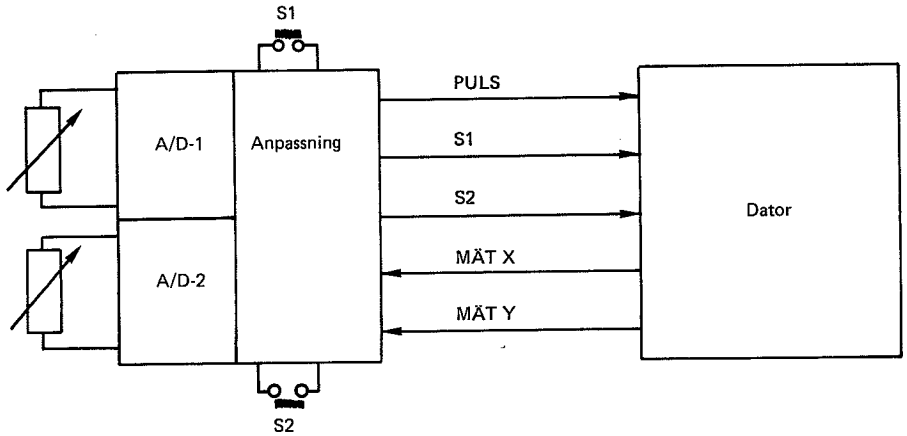


Fig 7.5

När vi beordrar en mätning med MÄT-X (dvs. när vi triggar monovippa 1) kommer vi på PULS att få en puls vars längd beror på X-potentiometern, dvs. spakens läge i X-led. Så länge pulsen pågår kommer signalen på denna tråd att vara "etta".

Genom att mäta denna tid, och sedan göra om proceduren för Y-potentiometern (MÄT-Y etc.) får vi två numeriska värden som entydigt bestämmer spakens läge.

De två tryckknapparna S1 och S2 är kopplade till varsin bit i V24-porten. När en knapp trycks in sätts motsvarande bit till "etta".

Det enda som återstår nu är att utföra signalanpassningen. V24-porten lämnar ifrån sig (och vill ta emot) -12 V som "etta" och +12 V som "nolla", medan TTL-kretsen 74LS123 arbetar med +5 V resp. 0 V.

Utan att gå in närmare på funktionen hos denna del av JOYSTICKen ges nedan ett kretsschema.

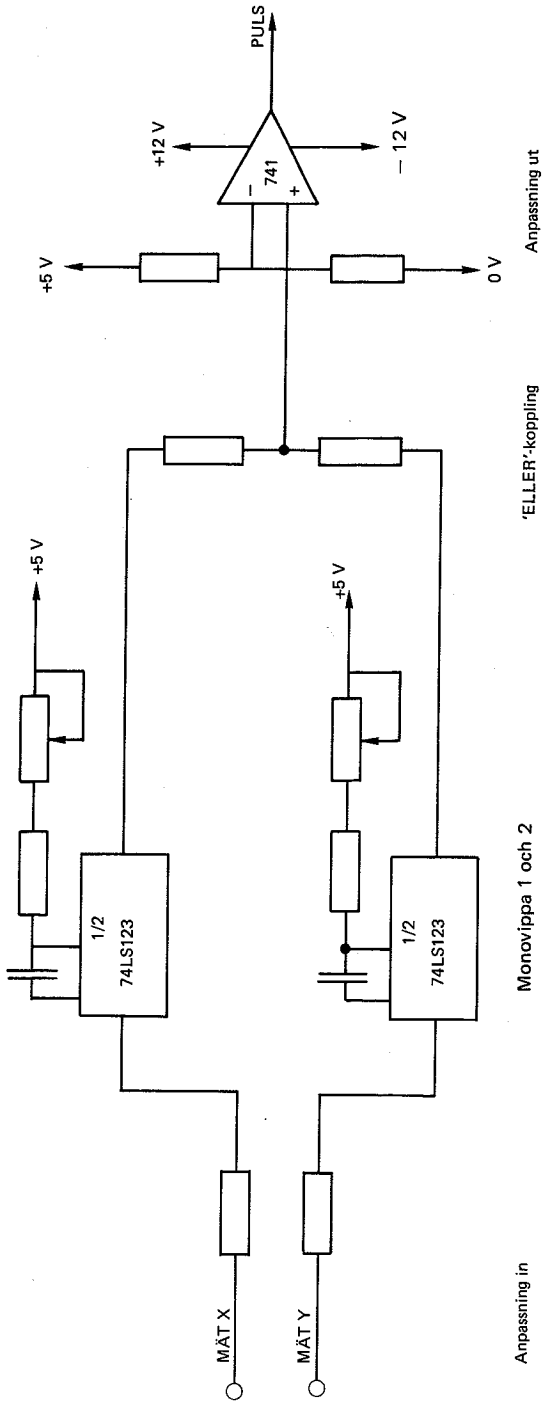


Fig 7.6

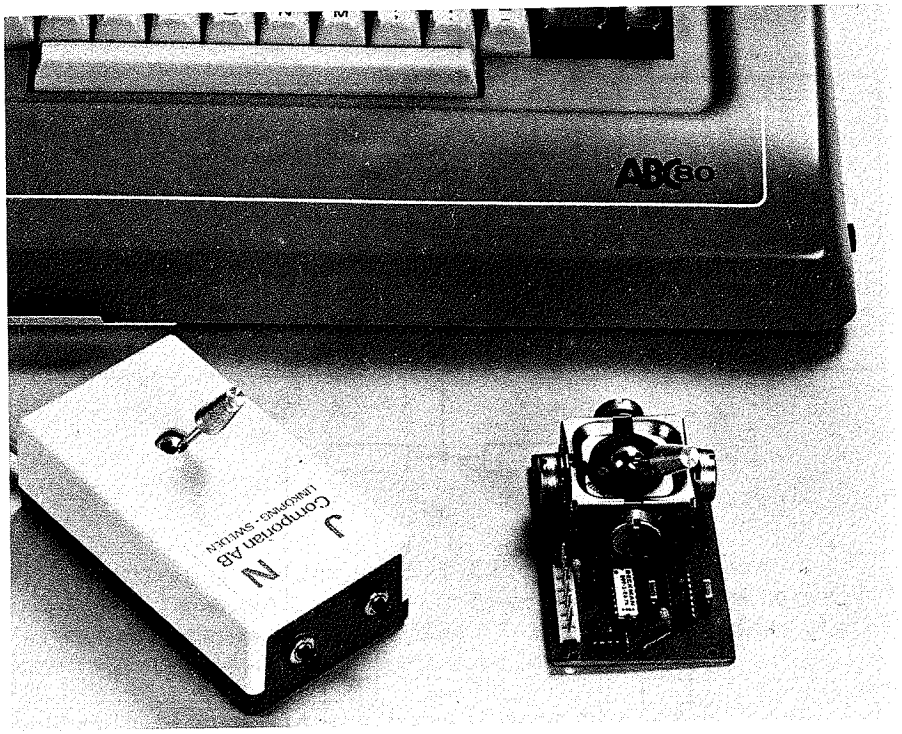


Fig 7.7
En färdigmonterad joystick.

7.3 Programmässig anslutning

JOYSTICKen kan anslutas till ABC80´s V24-kontakt (I/O - port nummer 58) enligt följande:

I/O-port 58

Bit 7 ...						... Bit 0
---	---	---	MÄT Y	MÄT X	S1	S2	PULS
Användes ej			ut	ut	in	in	in
Från/till ABC80							

MÄT-X och MÄT-Y är de signaler som beordrar mätning av X, resp. Y-koordinaten. S1 och S2 är insignaler från tryckknapparna. PULS slutligen är den bit där vi får in pulsen från JOYSTICKen. Denna bit är "etta" så länge som pulsen pågår, och "nolla" annars.

För att beordra mätning av X-koordinaten (Y-koordinaten) måste vi skicka ut en "nolla" följt av en "etta" på MÄT-X (MÄT-Y), trigga monovipporna.

```
1010 OUT 58,0 : OUT 58,8 : REM MÄT-X
```

Sedan ska programmet mäta hur lång pulsen är (dvs. hur länge det dröjer innan bit 0 blir "nolla" igen). Det kan göras så här:

```
1015 X%=0%
1017 X%=X%+1%
1020 IF (INP(58%) AND 1%)=1% THEN 1017
```

Men, för att få loopen så snabb som möjligt (vi vill ju inte vänta mer än nödvändigt på ett mätvärde) tillåter vi oss att använda en lite knepigare metod:

```
(1015 behövs ej)
1017 FOR X%=0% TO 32767%
1020 IF (INP(58%) AND 1%)=1% THEN NEXT X%
```

Så länge bit 0 är "etta" kommer NEXT X% att utföras - och X% räknas alltså upp - men när bit 0 blir "nolla" igen (pulsen är slut) fortsätter programmet med nästa rad. X% innehåller då ett tal som motsvarar stickans läge i X-led.

FOR-loopen går betydligt snabbare än GOTO-varianten, så vi behöver inte vänta lika länge för att få samma upplösning (givetvis måste pulslängden från JOYSTICKen justeras också).

De värden vi får behöver förmodligen skalas för att passa vårt ändamål. Om vi ska använda värdena för att placera markören t.ex, ska X-värdet variera mellan 0 och 23 och Y-värdet mellan 0 och 39, men om vi vill rita i grafisk mod blir motsvarande värden 0 - 70 resp. 2 - 79.

Denna skalning går till så, att vi tar reda på det största och det minsta X-värdet, kalla dem X0% resp. X9%. Detta kan vi göra en gång för alla.

Dessa värden använder vi sedan i vårt program. Om vi vill ha X-värden mellan 0 och N1 skriver vi:

```
10 XO%=13% : X9%=144% : REM Våra uppmätta värden
20 N1=23 : REM För textmod. Notera flyttal.
30 X1=N1/(X9%-XO%) : REM Notera flyttal.
.
.
1050 X%=(X%-XO%)*X1 : REM Skala X
```

Sedan gör vi samma sak med Y-värdet:

```
40 YO%=13% : Y9%=144%
50 N2=39 : REM Textmod
60 Y1=N2/(Y9%-YO%)
.
.
1060 Y%=(Y%-YO%)*Y1 : REM Skala Y
```

Ett litet program som läser av JOYSTICKen och sätter en markör på motsvarande punkt på skärmen kan då se ut så här:

```
10 REM *hpprog-seq*
20 REM Initiera
30 XO%=13% : X9%=144% : REM (våra uppmätta värden)
40 N1=23 : REM För textmod. Notera flyttal.
50 X1=N1/(X9%-XO%) : REM Notera flyttal
60 YO%=13% : Y9%=144% : X1%=0% : Y1%=0%
70 N2=39 : REM Textmod
80 Y1=N2/(Y9%-YO%)
90 ; CHR$(12);
100 REM *markör-ite*
110 GOSUB 230 : REM *mät*
120 IF R%=1% THEN 130
130 ; CUR(X1%,Y1%);´ ´; : REM Sudda gamla markören
140 ; CUR(X%,Y%);´¦´; : REM Sätt nya markören
150 X1%=X% : Y1%=Y% : REM Kom ihåg koordinaterna
160
170
180 GOTO 100
190 REM *markör-end*
200 REM *hpprog-end*
210 END
220 REM ------------------------------
230 REM *mät*
240 REM Subrutin som läser in X% och Y% från
250 REM JOYSTICKen.
260 OUT 58%,0% : OUT 58%,8% : REM Mät X
```



```
270 FOR X%=0% TO 32767%
280 IF (INP(58%) AND 1%)=1% THEN NEXT X%
290 OUT 58%,0% : OUT 58%,16% : REM Mät Y
300 FOR Y%=0% TO 32767%
310 IF (INP(58%) AND 1%)=1% THEN NEXT Y%
320 X%=(X%-XO%)/X1 : REM Justera X
330 Y%=(Y%-YO%)/Y1 : REM Justera Y
340 REM *mät-end*
350 RETURN
```

Vi kan också använda oss av tryckknapparna för att t ex rita på skärmen. Vi lägger till några rader i programmet:

```
105 IF R%=1% THEN 110
.
.
160 IF (INP(58%) AND 2%)=2% THEN R%=1% : REM S1
170 IF (INP(58%) AND 4%)=4% THEN R%=0% : REM S2
```

Om vi trycker på S1 kommer härefter markören inte att suddas när vi flyttar den, utan den ritar en linje. När vi trycker på S2 kommer markören i stället att sudda alla positioner den passerar.

När vi rör spaken fort märker vi att markören flyttar sig "hoppigt". Detta beror på att vi inte hinner mäta tillräckligt ofta - räkneloopen i BASIC tar ganska lång tid (lång i datorsammanhang: flera tiotals millisekunder).

Detta kan vi klara upp genom att skriva räkneloopen i maskinkod i stället för i BASIC. Maskinkodsprogrammet går mycket fortare än BASIC-programmet, så vi kan använda kortare pulslängder och ändå få samma upplösning, fast på mycket kortare tid. I kap. 9 ska vi titta närmare på det alternativet.

KAPITEL 8

BASIC-interpretatorn

ABC80

8.1 Tolken

Varför skulle man behöva veta något om Basic-tolken?

Ja, det kan kanske vara en berättigad fråga. För att skriva och köra BASIC-program behöver man ju inte alls veta vad som egentligen sker inne i maskinen, men det kan heller inte skada om man har lite hum om vad som händer i stort.

Ska man dessutom använda sig av större eller mindre maskinkodsrutiner (som anropas med CALL), kan det vara bra att veta var våra BASIC-variabler (A$, Q0% etc.) finns, var systemvariabler ligger (och vad de innehåller) etc.

Låt oss börja med att titta lite närmare på BASIC-TOLKEN, dvs. det maskinkodsprogram som startar när vi slår på spänningen, och som TOLKAR allt vi skriver in i ABC80.

Vad innebär det att det är en TOLK (interpretator) ? Jo, eftersom en mikroprocessor (som är den centrala delen i ABC80) inte kan förstå annat än program skrivna i maskinkod, och vi människor helst uttrycker oss på annat sätt, så läser tolken in våra BASIC-satser, och tolkar dem. Den översätter alltså inte BASIC-satserna till maskinkod, utan beroende på vilken sats (eller vilket kommando) vi har skrivit in, så hoppar tolken till någon lämplig subrutin som utför det vi begärde.

Men, det kan ju inte finnas en subrutin för varje tänkbar BASIC-sats, så först delas satsen upp i mindre delar - elementära operationer. Dessa kan vara: addera två tal, skriv ut ett tal, läs in en sträng från tangentbordet, och så vidare. Och för alla dessa finns speciella subrutiner i BASIC-tolken.

För att göra livet lättare för sig (och kanske framför allt för att göra exekveringen av program snabbare) omvandlar tolken våra inmatade satser till en INTERNKOD (pseudokod). Vitsen med det kan vi se i ett exempel:

```
10 FOR I=1 TO 1000
20 RANDOMIZE : PRINT RND;
30 NEXT I
```

Om texten lagrades precis som vi skrev in den, skulle "RANDOMIZE" på rad 20 ta 9 byte i minnet, och tolken skulle behöva jämföra alla

dessa 9 byte mot en tabell över tillåtna BASIC-ord en gång VARJE VARV i loopen, dvs. 1000 gånger i det här fallet. Sånt tar tid!

I stället jämför tolken "RANDOMIZE" mot tabellen EN gång - när vi skriver in uttrycket, och byter ut dessa 9 bytes mot en enda unik byte som betyder "RANDOMIZE".

Minnesbehovet blir mycket mindre, och det bästa av allt: när tolken stöter på denna byte som betyder "RANDOMIZE" vet den ju direkt vad den ska göra. Inget letande i tabellen när vi kör programmet alltså!

Dessutom görs en extra genomgång av programmet när vi skriver "RUN". Alla "GOTO" och "GOSUB" etc. som refererar ett radnummer ändras så att de direkt refererar den adress i minnet där raden ligger. Även variabelreferenser fixas så att de pekar direkt på den adress i minnet där variabelns värde finns.

Det är denna SEMI-KOMPILERING (omvandling av BASIC-ord till unika bytar, och lite annan transformering av uttrycken) som gör ABC80´s BASIC så snabb.

LIST-rutinen vet om alla dessa transformationer, och gör motsatsen när vi begär listning av ett program, antingen vi listar programmet på skärmen, eller ut på en fil.

Om vi däremot sparar programmet med "SAVE", sparas den transformerade koden precis som den ligger i minnet. En sådan fil går alltså snabbare att läsa in igen, än ett program som är sparat med "LIST" eftersom det senare ju måste transformeras när det läses in.

8.2 Symboltabellen

Alla de variabler - heltal, flyttal och strängar - som vi använder oss av i vårt BASIC-program, samlas upp i en tabell som ligger direkt efter programmet i minnet.

Denna tabell skapas av tolken först när vi ger "RUN"-kommandot och i den lagras information (för tolken) om variablerna, och reserveras plats för deras värden. (vi bortser från hur det går till när vi gör t ex LET A=7 i kommandomod),

En systemvariabel - (VARIABELROTEN på adress 65065) - pekar på symboltabellens början, och sen är hela tabellen en LÄNKAD LISTA, dvs. varje post i tabellen pekar vidare till nästa post. Tack vare detta behöver inte alla poster vara lika långa. Den allra sista posten har en nästa-pekare som är noll, vilket betyder slutet på tabellen.

En schematisk figur över symboltabellen skulle kunna se ut så här:

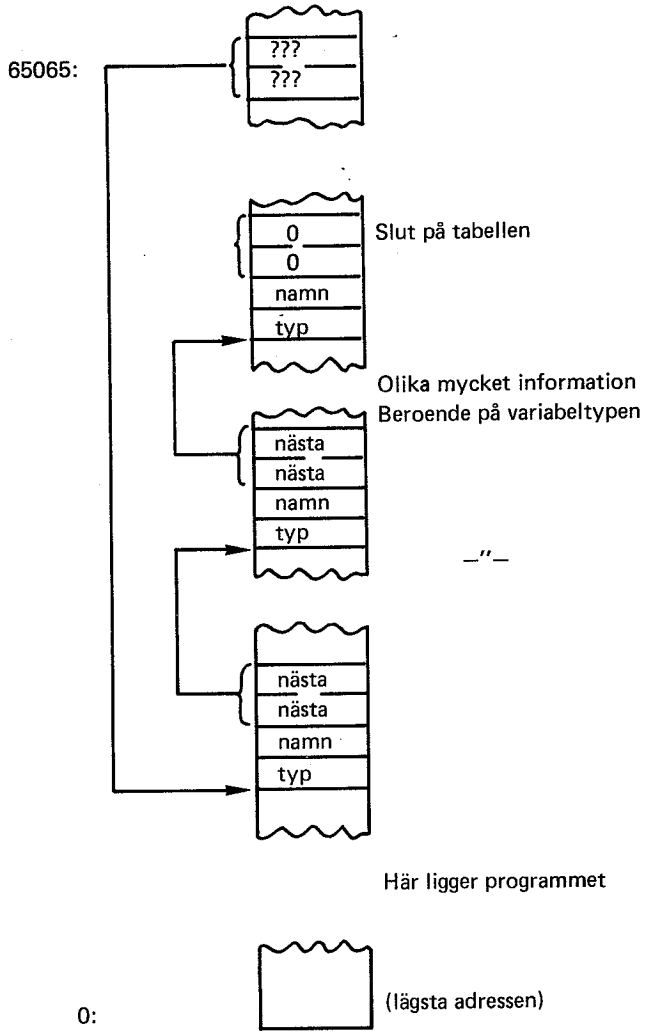


Fig 8.1 Principen för symboltabellen.

(Variabelroten (65065) pekar på början)

De två första byten i varje post innehåller alltså variabelns typ och namn enligt nedanstående:

Byte 1: siffra,typ Byte 2: bokstav

Som vi ser är den första byten delad i två delar. De första fyra bitarna står för den siffra som kan ingå i variabelnamnet (Ex. 3 i A3%) och är enbart ettor (F hex) om variabelnamnet inte innehåller någon siffra alls (som t.ex G$). De sista fyra bitarna betecknar variabelns typ, och har följande betydelse:

Binärt	Hex	Typ
0000	0	Flyttal
0001	1	Heltal
0010	2	Sträng
0011	3	-----
0100	4	Flyttalsvektor
0101	5	Heltalsvektor
0110	6	Strängvektor
0111	7	----
1000	8	Flyttalsmatris
1001	9	Heltalsmatris
1010	A	Strängmatris
1011-1111	B-F	----

Exempel på hur variabler betecknas i tabellen:

Flyttalet A	F0 41	(ingen siffra, flyttal) "A"
Heltalet B3%	31 42	(siffran 3, heltal) "B"
Strängmatrisen C9$(?,?)	9A 43	(siffran 9, strängmatris) "C"

etc.

Efter nästa-pekaren kommer så den variabelberoende informationen - variabelns värde, storlek på strängar och matriser etc. - enligt följande:

Flyttal	S S S T E
Heltal	V V
Sträng	D D R R L L
Flytt. vekt.	A A M M T T
Helt. vekt.	A A M M T T
Sträng vekt.	Q Q M M T T
Flytt. matr.	A A M M T T X X Y Y
Helt. matr.	A A M M T T X X Y Y
Sträng matr.	Q Q M M T T X X Y Y

Där

V V är ett heltalsvärde (swappade bytar).

SSSTE är ett flyttalsvärde (S=två BCDsiffror, T=tecken och E=10 exponent+128).

A A är adress till vektorns (matrisens) början.

M M är det totala antalet element i en vektor (matris).

X X är övre gräns för det första indexet i en matris.

Y Y är övre gräns för det andra indexet i en matris.

D D är dimensionerad längd på strängen.

L L är aktuell längd på strängen.

R R är adressen till strängens värde (dvs pekar på den plats i minnet där tecknen som ingår i strängen ligger).

Q Q är pekare till en tabell över alla enskilda strängar i en strängvektor (matris). Denna tabell består av en uppsättning D D R R L L för varje element i vektorn (matrisen).

T T Används av BASIC-tolken för tillfällig lagring av mellanresultat under beräkningen.

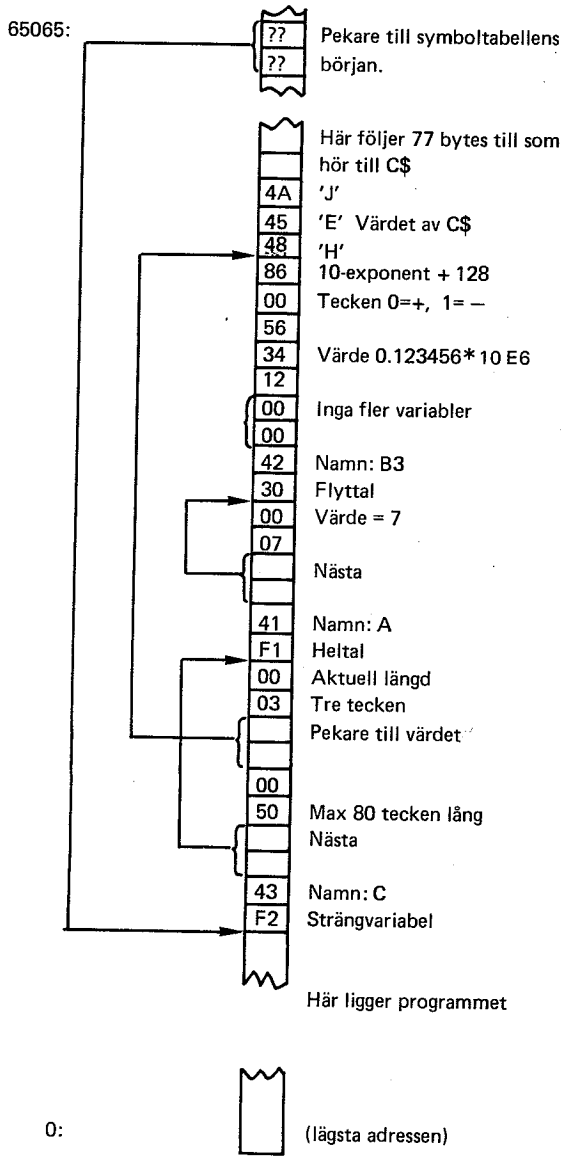


Fig 8.2 Symboltabell. Ett exempel.

Alla matriser ligger lagrade så att det andra indexet varierar snabbast (radvis).

Ett litet exempel:

```
10 C$="HEJ"
20 A%=7%
30 B3=123456
40 STOP
```

Detta program ger upphov till symboltabellen i fig. 8.2 (siffror i hex).

Vad ska vi nu med den här kunskapen till? Jo, när (om) vi vill blanda BASIC-program och maskinkodsrutiner behöver vi oftast ha någon form av kommunikation mellan BASIC-programmet och maskinkodsrutinen. Med kännedom om symboltabellens utseende kan vi få en maskinkodsrutin att leta reda på adressen till, och ändra värdet på t ex variabeln Q0$ eller vilken annan variabel som helst (precis som sker när vi använder CALL för direktläsning på fil).

8.3 Systemvariabler

Precis som våra BASIC-program behöver variabler (A$, Q0% etc.), så behöver tolken det. Den måste ju hålla reda på var i BASIC-programmet den håller på att tolka, vad klockan är, vilka filer som är öppnade, var på skärmen markören håller hus, och så vidare.

Dessa variabler (systemvariablerna) ligger samlade i minnet mellan adresserna 64768 och 65407. En del av dem har vi stött på tidigare: realtidsklockan t.ex, och blocknummercellen (65021) som vi utnyttjade i kapitel 4 vid skrivning på kassett.

Det finns flera systemvariabler som kan vara användbara i vissa fall. Vi kan läsa dem med PEEK, och ändra deras värde med POKE (Se upp noga! Ändrar vi fel systemvariabel kan ABC80 uppföra sig väldigt konstigt).

Här följer en kort sammanfattning av några (mer eller mindre) användbara variabler.

Där två adresser anges gäller det ett helt block med information.

Om adressen är märkt med "B" så är det en byte det gäller, och värdet fås med PEEK(adress). Om den är märkt med "W" är det ett heltal, vars värde fås med PEEK(adress) + SWAP%(PEEK(adress + 1)).

ADRESS		Användning
64768 - 65007		Används huvudsakligen av DOS-et (Flexskive-hanteringen). Dessa celler ska vi akta oss noga för att skriva något i. Man vet aldrig vad som kan hända med skivorna annars.
65008 - 65010		Realtidsklockan. Se särskilt avsnitt.
65011	B	Innehåller numret på den rad markören just nu befinner sig på.
65012	B	Innehåller kolumnnumret för markören. Genom att läsa denna och förgående cell, kan vi få reda på var markören befinner sig. Att skriva något i dem är ekvivalent med att använda funktionen CUR(X,Y). Ex. POKE 65011,12,13 ger samma resultat som PRINT CUR(12,13).
65013	B	Bit 7=1 signalerar att en tangent har blivit nertryckt på tangentbordet. Om PEEK(65013)AND 128 är skilt ifrån noll, så har en tangent tryckts ner. Biten återställs till noll när GET, INPUT eller INPUTLINE utförs.
65021	B	Innehåller nummer på senast utskrivna kassettblock. Se kapitel 4.
65031	B	Bit 7=1 anger att CTRL-C har skrivits in. Kan inte utnyttjas från BASIC, men kan vara användbart i maskinkodsrutiner.
65034	W	Pekar på ENHETSLISTAN. Se särskilt avsnitt.
65052	W	BOFA. Pekar på programbörjan. Sätts automatiskt till lägsta möjliga adress vid "RESET". Genom att ändra dessa celler och göra "NEW" kan vi "gömma"

en del av minnet, t ex för att ha maskinkodsrutiner i.

65054 W — EOFA. Pekar på sista byten av det BASIC-program som ligger i minnet just nu.

65056 W — HEAP. Pekar på första lediga minnescell efter program + symboltabell. Vi kan dock inte utnyttja denna delen av minnet, eftersom det är här som BASIC-tolken lägger mellanresultat under pågående beräkningar.

65063 W — Pekar på högsta tillåtna adress för BASIC-program. Även dessa celler kan vi ändra och få ledigt utrymme för POKE. Jämför adress 65054, och ABC80´s bruksanvisning (sid 58 - 59).

65065 W — Det här är variabelroten, nämd i föregående avsnitt.

65067 B — Innehåller nummer på senast utvalda I/O-kort.

65074 W — Pekare till lista över öppna filer.

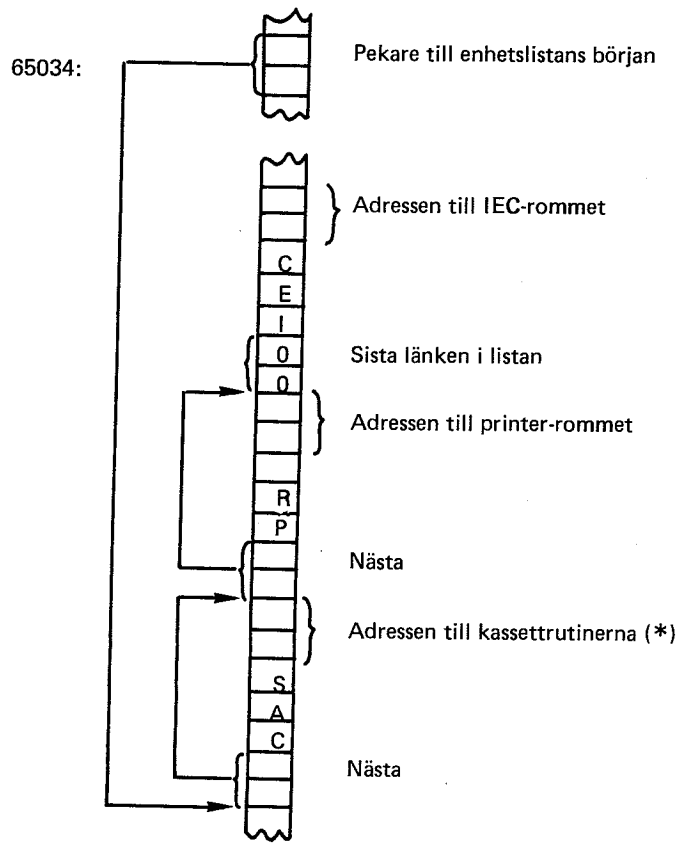
65088 - 65207 — Inmatningsbuffert. Här hamnar först allt vi skriver in i ABC80.

65208 - 65328 — Editeringsbuffert och slaskarea vid transformering till internkod.

65408 - 65535 — Ledig area (om vi inte har printeroptionen ansluten, då är det radbuffert för printern).

8.4 Enhetslistan

För att strukturera och förenkla in och utmatning har ENHETS-begreppet införts i ACB80 (ex. på enheter är CAS:, PR:, DRO: etc.). Detta innebär att BASIC-tolken inte behöver bekymra sig om hur in och utmatning egentligen sker på en viss enhet, utan bara vet en adress att hoppa till när t.ex utmatning av ett tecken ska ske.




```
(*)
CASRUT  JP  COPEN    ;  Rutin för att öppna fil
        JP  CPREP    ;  För att göra 'PREPARE'
        JP  CCLOSE   ;  Stänger en fil
        JP  CPRINT   ;  Skriver på fil
        JP  CINPUT   ;  Läser från fil
        ...
```

Fig 8.3 Enhetstabellen (utan diskettenhet).

För att detta ska fungera måste enheten vara definierad, och en maskinkodsrutin som gör den aktuella in/utmatningen måste finnas i minnet.

Några enheter finns alltid definierade i ABC80, nämligen CAS:, som är kassetten, och DR0: och DR1: om vi har floppy disk ansluten. Bildskärmen och tangentbordet är speciella, i det att kommunikationen med dem INTE går via enhetslistan, utan sköts direkt av BASIC-tolken. Filnummer noll (0) anger tangentbord (INPUT #0 ..) och bildskärm (PRINT #0 ..).

En drivrutin för en enhet skall alltid börja på ett standardiserat sätt: en uppsättning hopp till olika delar inom drivrutinen, som utför de olika operationerna på enheten (OPEN, PREPARE, CLOSE, INPUT och PRINT är de viktigaste).

I och med detta behöver BASIC-tolken bara känna till den allra första adressen för drivrutinen, hoppen till de olika delarna ligger ju alltid på ett bestämt avstånd från denna startadress.

Startadresserna och enheternas namn finns lagrat i ENHETSLISTAN. Det är en länkad lista (på samma sätt som symboltabellen), och behöver därför inte ligga samlad på ett ställe i minnet. En systemvariabel (65034 - 65035) pekar på den första länken i enhetslistan, och noll som "nästa - pekare" betyder att listan är slut.

8.5 Realtidsklockan

ABC80 har som bekant en realtidsklocka. Den räknas ner en gång var tjugonde millisekund, så länge strömmen är påslagen. Dess värde ligger lagrat i tre bytes i minnet, på adresserna 65008 - 65010. Den minst signifikanta byten (den som räknas ner oftast) ligger på adressen 65008.

När vi vill använda oss av klockan i ett BASIC-program behöver vi bara läsa dessa celler, och sedan omvandla värdena till klockslag.

Exempel.

```
1000 K%(0%)=PEEK(65008%)
1010 K%(1%)=PEEK(65009%)
1020 K%(2%)=PEEK(65010%)
```

Denna lilla programsnutt plockar upp klockans tre byte i vektorn K%. Här kan dock ett problem tillstöta: Vad händer om klockan ändrar sig medan vi håller på att läsa av den?

Säg att vi har värdena 1, 0, och 1 i de tre cellerna när loopen startar. Vi läser den första och den andra byten till K%(0) och K%(1), och just då är det dags för klockan att slå om. Efter det har vi 0, 255, 0 i de tre cellerna (se nedan för närmare förklaring av detta faktum !) när det blir dags att läsa den sista cellen, och vi får det helt felaktiga värdet 1, 0, 0 i K%(0), K%(1) resp. K%(2)!

För att säkra oss mot detta måste vi kontrollera att inte klockan har ändrats medan vi läste den. Det kan vi göra genom att se efter om det står samma sak i den snabbaste byten både före och efter det att vi har läst av klockan. Det kan vi göra så här (t ex):

```
1030 IF PEEK(65008%)<>K%(0%) THEN 1000
```

Dvs. om värdet har ändrats medan vi läste så får vi vackert göra om alltihop från början (det är ju inte så fasligt mycket att göra om, det tar inte särskilt lång tid).

Ovan lägger vi också märke till en annan egenhet hos realtidsklockan: Byte 2 minskas med 1 när byte 1 blir NOLL, inte när den ändras från 0 till 255. Detta är en följd av att avbrottsrutinen som räknar ner klockan ska vara så kort som möjligt. Det innebär dock att vi måste tänka oss för en gång extra, när vi ska använda oss av klockan. Vi måste komma ihåg att byte 1 (den som ändras snabbast) hela tiden är en enhet för stor, och att 0 - 1 = 255 när vi räknar med byte.

Om vi inte tar hänsyn till den här egenheten kan det ibland se ut som om klockan "går baklänges", och det är ju inte vidare lyckat.

Rutinerna för att ställa och läsa av klockan som finns beskrivna i bruksanvisningen till ABC80 (sid. 59 - 60), bör därför justeras med hänsyn taget till felet i byte 1. Med samma radnummer som i bruksanvisningen blir ändringarna:

```
1055 Z%=(Z%+1%) AND 255% : REM Ny rad
```

I rutinen som ställer klockan, och:

```
2040 Z%=PEEK(T1%)
2045 Z1%=PEEK(T1%+1%) XOR 255%
2050 Z2%=PEEK(T1%+2%) XOR 255%
2055 IF Z%<>PEEK(T1%) THEN 2040
2060 Z%=((Z%-1%) AND 255%) XOR 255%
2070 Z=Z%/50+5.12*(Z1%*256+Z2%)
```

I rutinen som läser av den.

KAPITEL 9

Assembler-programmering

ABC80

9.1 Varför använda assemblerprogram?

Assembler användes när man kräver full kontroll över datorns beteende, exempelvis om en viss rutin måste exekvera väldigt snabbt eller om man ska göra komplicerade in/ut operationer, eventuellt med avbrott. Full kontroll betyder bland annat att man kan kringå onödiga delar i BASIC mm som slöar ner ett program. Förutom att man kan skriva snabba rutiner har man ibland möjlighet att minska kodens storlek.

Exempel på program skrivna i assembler är:

1. BASIC tolken
   Det program som tolkar BASICprogram är av naturliga skäl skrivet i assembler.

2. Schack
   Det finns ett schackspelande program för ABC80 som av effektivitetsskäl (snabbhet) är skrivet i assembler.

9.2 BASIC kontra assemblerkod

1. BASIC är långsammare än maskinkod på grund av att arbetet som utförs för att tolka ett BASICprogram är ganska omfattande.

2. BASIC är kompaktare än maskinkod på grund av att en BASICsats motsvarar flera (många) maskinkods satser och alltså innehåller mer information.

3. Man har inte tillgång till alla maskinens resurser från BASICen men har inga sådana problem med maskinkod.

9.3 Hur använder man assemblerprogram?

9.3.1 ASSEMBLERPROGRAM MÅSTE ASSEMBLERAS

För att kunna köra ett assemblerprogram måste man översätta det till en form som ABC80 förstår, detta sker med hjälp av en assemblator. Varje sats i assemblerprogrammet översätts till en 1-4 byte lång maskininstruktion eller tolkas som ett kommando till assemblatorn. (Motsvarande översättning i BASIC sker varje gång man exekverar en sats.) Detta innebär att vi för varje gång vi ändrar i vårt program måste göra om hela översättningen innan det är körbart i sin nya version.

Ett annat sätt att översätta ett assemblerprogram är att göra det för hand. Det innebär att man skriver sitt program på vanligt sätt och sedan med hjälp av Zilogs opkodstabeller översätter det byte för byte. Metoden är ganska tidskrävande och lämpar sig bara för små program, svårigheter kan uppstå när man måste beräkna relativa adresser för hand.

9.3.2 ETT ASSEMBLERPROGRAMS FORM

Ett assemblerprogram bör bestå av:

1. Programhuvud
   Programmets namn och relevanta kommentarer.

2. Deklarationer av konstanter
   Här deklareras konstanter som ska användas av assemblatorn.

3. Program
   Programsatser blandade med kommandon till assemblatorn.

4. Data
   Här läggs variabler och konstanter som måste finnas när programmet körs.

5. Fler instanser av typ 3 och 4.

6. Programslut
   Består av direktivet END.

Detta är den rekommenderade strukturen hos ett assemblerprogram, den har inte så mycket med dess funktion att göra men väl läsbarheten.

9.4 Några nyttiga begrepp

Assemblerprogram rör sig i vissa stycken med begrepp som inte har någon motsvarighet i BASIC, här följer en kort presentation av några av dem.

9.4.1 BITAR OCH BYTES

En normal dator manipulerar binära data. Detta innebär att den minsta informationsenheten (en bit) kan bara ha ett av två värden, 1/0, JA/NEJ, PÅ/AV. För att förenkla hanteringen tillåter man i allmänhet operationer på sekvenser av bitar. Z80 kan arbeta på 16bit, 8bit (1 byte), 4bit (1 nybble) eller 1 bit enheter.

9.4.2 REGISTER

Ett register är en minnescell som finns internt i en dator, det vanliga minnet betraktas i detta sammanhang som en yttre enhet. Fördelar med register är att det går snabbt att skriva och läsa i dem (kort accesstid) och att det är enkelt att adressera dem. Ett stort antal instruktioner i Z80 kan dessutom bara operera på data i olika register. Z80´s registeruppsättning består av två grupper med åtta 8bit register i varje plus fyra 16bit register samt två specialregister (se fig.).

1. A,F,B,C,D,E,H,L
   finns i två uppsättningar.
   A
   är huvudackumulatorn, här sker alla aritmetiska operationer.

   F
   här hamnar de flaggor som visar resultatet av vissa operationer.

   BC, DE, HL
   användes som 16bit register eller individuellt som 6 st 8bit register (B,C,D,E,H,L). HL är speciellt viktigt som adressregister.

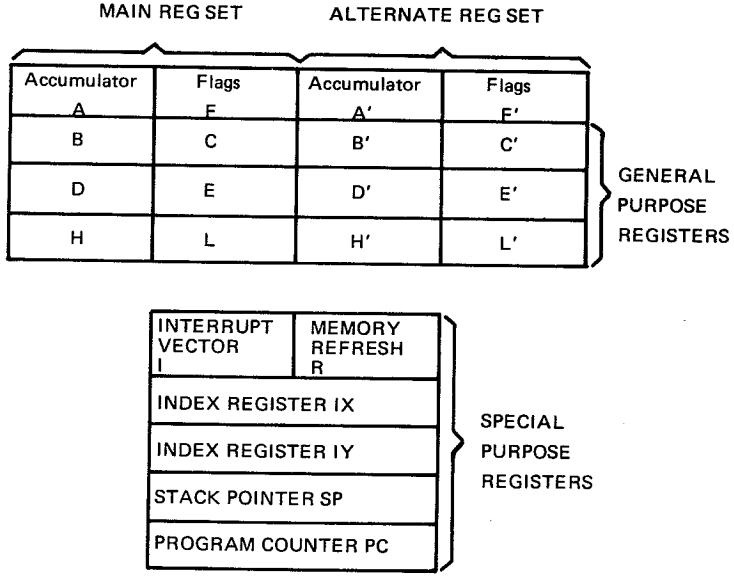
2. I
   Användes för avbrottshantering (mod 2).

3. R
   Användes av refresh-mekanismen, kan knappast användas till något annat utom möjligen som slumptals-generator.

4. IX,IY
   Det här är indexregistren, användes för adressberäkningar när man har begärt indexerad adressering.

5. SP
   stackpekaren användes vid subrutinhantering och avbrott mm.

6. PC
   Programräknaren, pekar på den instruktion som kommer att utföras härnäst.



Z-80 CPU REGISTER CONFIGURATION

Fig 9.1

9.4.3 FLAGGOR

En bit som användes för att markera ett tillstånd (t ex om en addition lämnat ett felaktigt resultat) kallas för en flagga. Z80 har ett register (F) som bara innehåller flaggor.
Observera att somliga instruktioner sätter flaggorna, andra inte.

Man måste veta hur dom beter sig eftersom det inte alltid är så konsekvent att man kan räkna ut det.
Flaggorna är:

1. C (carry)
   Anger om den närmast föregående aritmetiska operationen gav en minnessiffra.

2. Z (zero)
   Anger om en operations resultat var lika med noll.

3. S (sign)
   Anger om en operation gav ett negativt resultat.

4. P/V (parity/overflow)
   Anger ibland om det blev overflow (anger om resultatet av en aritmetisk operation blev så stort att det inte rymms i den cell det ska lagras i, 120+120 = 240 vilket är större än 127; max positivt tal i en byte), ibland vilken paritet resultatet hade.

5. H (half carry)
   är minnessiffran vid BCD operationer, alltså från bit 3.

6. N (subtract)
   Användes av DAA instruktionen för att göra en korrekt decimal justering.

9.4.4 ADRESSERING

För att peka ut den minnescell man vill manipulera med en instruktion anger man en adress. En adress kan även vara ett registernamn. För att bilda en adress till en minnescell använder man en av de adresseringsmetoder som Z80 erbjuder:

1. Immediate (omedelbar) kort/lång
   Detta innebär att man hämtar data från cellen efter instruktionen. Adressen blir alltså PC+n om n är instruktionens längd. Det finns möjlighet att hämta både en och två bytes vilket benämns kort/lång operand.

2. Relativ
   Användes av vissa hoppinstruktioner för att medge adressering av instruktionens närmaste omgivning. Ett hopp kan göras till positioner +129 till -126 bytes från nuvarande läge med en 1 byte lång adress.

3. Extended (direkt)
   En 16bit adress som direkt pekar ut önskad position i minnet.

4. Indexed (indexerad)
   Om man formar adressen genom att till en direkt adress (som ligger i ett register t ex) addera ett index kan man hantera tabeller på ett effektivt sätt. I det här fallet ligger basadressen i register IX eller IY och indexet i en byte efter instruktionen. Indexets värde varierar alltså mellan +127 och -128 positioner, basregistrets innehåll påverkas inte av denna operation.

5. Register indirect (indirekt)
   Samma som direkt adressering fast adressen finns i ett register (typiskt HL).

En specialvariant av indexerad adressering är så kallad stackadressering. Detta innebär att man vid varje referens via stackpekaren (ett speciellt indexregister) automatiskt ändrar den. Detta sker i Z80 t ex genom att att använda POP och PUSH instruktionena. Vid en PUSH räknas SP först ner ett steg och användes sedan som indexregister; POP gör att man indexerar med SP och sedan räknar upp SP. Detta medför att stacken växer neråt.

När man ska ange vad som är en adress och vad som är innehållet i en adresserad cell använder man ofta parenteser för att markera att man menar innehållet.
Man slopar ibland parenteserna kring registernamn trots att man menar registrets innehåll.

9.5 Instruktionerna

På samma sätt som ett BASICprogram byggs upp av satser byggs ett assemblerprogram upp av instruktioner. Dessa instruktioner är olika för de flesta existerande datorer, för Z80 gäller följande:

9.5.1 DATAFÖRFLYTTNINGAR (LOAD and EXCHANGE)

Instruktionen LD användes för att flytta data mellan register och register/minne. Den kan flytta data i båda riktningarna och motsvarar instruktionerna LOAD, STORE och MOVE på diverse andra maskiner. I grundutförandet flyttar den ett (8bit) register till ett (8bit) register och skrives då som:

LD A,B

vilket flyttar innehållet i register B till register A. Man kan byta den ena (eller båda) registeradressen mot något av:

1. Register indirect LD A,(r)
   Ladda A med innehållet i den cell vars adress finns i r (r är en av: HL, BC eller DE).

2. Indexed LD A,(r+n)
   Ladda A med cell r+n´s innehåll. n är ett index i området -128 till +127 och r är en av IX eller IY.

3. Extended LD A,(n)
   Ladda A med cell n´s innehåll, n är en absolut 16bit adress.

4. Implied LD r,A
   Ladda register r med A (r är en av: R eller I).

5. Immediate LD A,n
   Ladda A med n.

Figuren nedan visar vilka kombinationer av käll- (source) och destinations-adresserna som är tillåtna. Exempel:

1. LD (IY + 17), 31H
   Ladda cellen med address (IY) + 17 med 31 (hexadecimalt).

2. LD E, (HL)
   Ladda register E med innehållet i cellen vars adress står i HL.

SOURCE

Destination		IMPLIED I	IMPLIED R	REGISTER A	B	C	D	E	H	L	REG INDIRECT (HL)	(BC)	(DE)	INDEXED (IX+d)	(IY+d)	Ext. Addr. (nn)	Imme. n
Register	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A n n	3E n
	B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
	C			4F	48	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
	D			57	50	51	52	53	54	55	56			DD 56 d	FD 56 d		16 n
	E			5F	58	59	5A	5B	5C	5D	5E			DD 5E d	FD 5E d		1E n
	H			67	60	61	62	63	64	65	66			DD 66 d	FD 66 d		26 n
	L			6F	68	69	6A	6B	6C	6D	6E			DD 6E	FD 6E		2E n
Reg Indirect	(HL)			77	70	71	72	73	74	75							36 n
	(BC)			02													
	(DE)			12													
Indexed	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
	(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
Ext.addr.	(nn)			32 n n													
Implied	I			ED 47													
	R			ED 4F													

8 BIT LOAD GROUP
'LD'

Fig 9.2

LD kan även hantera 16bit operander för att flytta registerpar till och från minnet. De rätt speciella POP och PUSH instruktionerna finns dessutom med i den här gruppen.
De kan hantera registerparen AF, BC, DE, HL, IX och IY.

Exempel:

Stacken användes ofta för att spara undan register tillfälligt. Observera att POP sekvensen sker omvänt mot PUSHarna.

```
rädda HL och AF
PUSH HL
PUSH AF
här användes HL och AF fritt
POP AF
POP HL
HL och AF är nu återställda.
```

I figuren ser vi vilka adresseringskombinationer som är tillåtna för de olika 16bit LD instruktionerna.

Det finns dessutom en EXchange instruktion som byter innehåll i ett register och ett register/minnescell.

1. EX (SP),HL
   Byt toppen av stacken och HL.

2. EX (SP),IX

3. EX (SP),IY

4. EX DE,HL
   Byt innehållet i DE med HL.

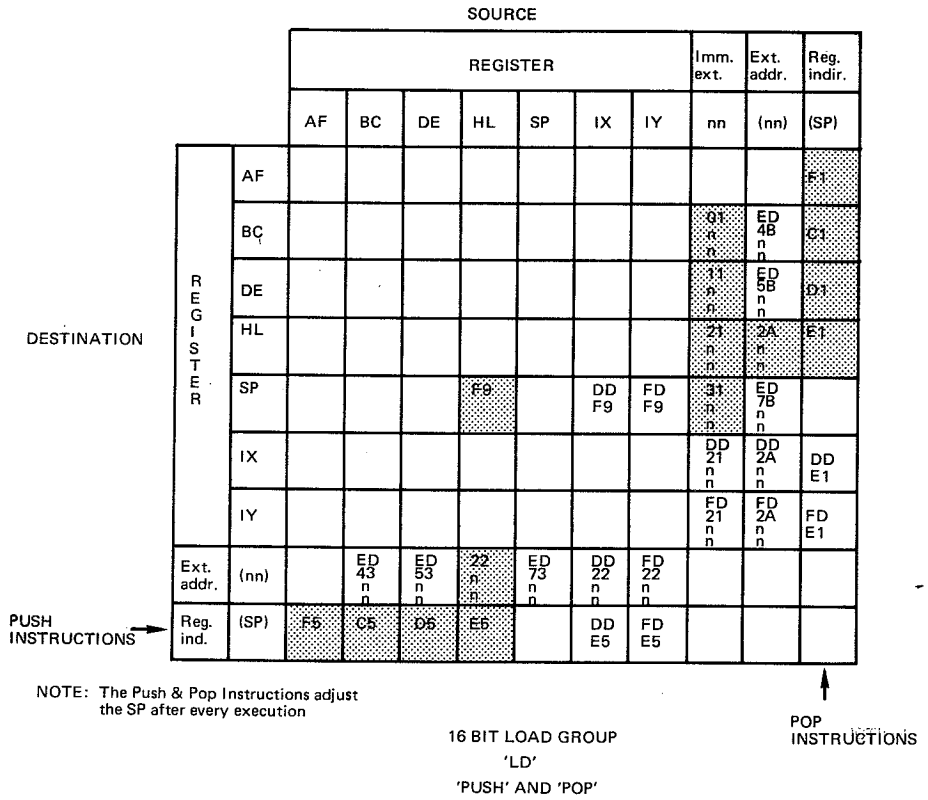


Fig 9.3

9.5.2 REGISTER GROUP EXCHANGE

För att kunna förflytta sig mellan de register som är dubblerade i Z80 finns det några "byt" instruktioner.
Registrens innehåll påverkas inte, man flyttar bara processorns uppmärksammhet från en uppsättning till en annan.

1. EXX
   Byt grupp BCDEHL mot B´C´D´E´H´L´.

2. EX AF,AF´
   Byt grupp AF mot AF´.
   Notera skillnaden mot de andra EX instruktionerna.
   Obs! Byter A och F mot A´ och F´

9.5.3 ARITHMETIC AND LOGICAL

Det finns ett antal instruktioner som utför aritmetiska och logiska operationer. De opererar på antingen 8 eller 16 bit data, man kan alltså göra en addition på t ex IX registret i en operation. För alla binära 8bit operationer i gruppen gäller att operationen utförs på värdet i ackumulatorn (A) och den cell man adresserat. Resultatet hamnar i A plus att flaggorna i F sätts. För INC och DEC gäller att resultatet hamnar i den adresserade cellen.
Tillåtna adresseringssätt är:

1. Register A,B,C,D,E,H,L

2. Register indirect (HL)

3. Indexed (IX+n), (IY+n)

4. Immediate n (ej för INC och DEC)

8bit instruktionerna är:

1. ADD x
   Addera binärt.

2. ADC x
   Addera binärt men tag med den carrybit (minnessiffra) som bildades vid den förra operationen.

3. SUB x
   Subtrahera.

4. SBC x
   Subtrahera med carry (som ADC).

5. AND x
   Gör en bitvis AND (och) operation.

6. XOR x
   Gör en bitvis XOR (exklusivt eller) operation.

7. OR x
   Gör en bitvis OR (eller) operation.

8. CP x
   Jämför A registret med operanden och sätt flaggorna i F i enlighet med resultatet. A påverkas inte.

9. INC x
   Addera ett till den adresserade cellen (eller registret).

10. DEC x
    Subtrahera ett från den adresserade cellen/registret.

SOURCE

DESTINATION		BC	DE	HL	SP	IX	IY
'ADD'	HL	09	19	29	39		
	IX	DD 09	DD 19		DD 39	DD 29	
	IY	FD 09	FD 19		FD 39		FD 29
Add with carry and set flags 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
Sub with carry and set flags 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
INCREMENT 'INC'		03	13	23	33	DD 23	FD 23
DECREMENT 'DEC'		0B	1B	2B	3B	DD 28	FD 28

16 BIT ARITHMETIC

Fig 9.4

Följande arbetar på 16bit data på liknande sätt som 8bit instruktionerna. Här måste man dock ange både käll- och destinations-adress.
Ett exempel per tillåten instruktion.

1. ADD IY,IY
   Multiplicera IY med två.

2. ADC HL,BC
   Addera BC (och carry biten) till HL.

3. SBC HL,SP
   Subtrahera SP från HL (använd carry biten).

4. INC IX
   Öka IX med ett.

5. DEC SP
   POPa ett element från stacken utan att hämta in det.

Figuren visar vilka adresseringskombinationer som är tillåtna för de aritmetiska 16bit instruktionerna.

Det finns dessutom några specialinstruktioner som opererar på ackumulatorn eller carry flaggan:

1. DAA Decimal Adjust accumulator
   Användes för BCD-aritmetik, fungerar efter både ADD och SUB.

2. CPL Complement accumulator
   Inverterar alla bitarna i ackumulatorn.

3. NEG Negate accumulator
   Negerar ackumulatorn.

4. CCF Complement Carry Flag
   Byt värde på carry biten.

5. SCF Set Carry Flag
   Sätt carry biten till ett.

9.5.4 ROTATE AND SHIFT

Man vill i en dator ha möjlighet att manipulera de enskilda bitarna

i minnet på ett effektivt sätt, detta sker i Z80 framför allt med skift och bit gruppernas instruktioner. Att skifta ett tal är exakt samma sak som att upprepade gånger multiplicera eller dividera (beroende på skiftriktning) med 2. Rotation innebär att man skiftar sitt tal i cirkel. Alla skift sker med ett steg per instruktion.
n är en av (HL), (IX+d), (IY+d), B, C, D, E, H, L eller A. Skift gruppens instruktioner är:

1. RLC n
   Rotera cellen åt vänster, kopiera minst signifikanta biten till carrybiten i F.

2. RL n
   Rotera carrybiten + cellen (de betraktas som sammanhängande, i den ordningen) åt vänster.

3. RRC n
   Rotera cellen åt höger, kopiera mest signifikanta biten till carrybiten.

4. RR n
   Som RL fast åt höger.

5. SLA n
   Skifta carry + cellen åt vänster, sätt minst signifikanta biten i cellen till noll.

6. SRA n
   Skifta cellen + carry åt höger, cellens mest signifikanta bit påverkas ej.

7. SRL n
   Skifta cellen + carry åt höger, cellens mest signifikanta bit sätts till noll.

De följande två instruktionerna arbetar med BCD siffror (4bit). De medger att man roterar nedre halvan av ackumulatorn och en minnescell, utpekad av HL, fyra steg.

8. RLD
   Rotera åt vänster.

9. RRD
   Rotera åt höger.

9.5.5 BIT MANIPULATING

Bitgruppen erbjuder goda möjligheter att hantera enskilda bitar i register och minne.
Tillåtna adresseringssätt är:

1. Register A,B,C,D,E,H,L

2. Register indirect (HL)

3. Indexed (IX+n), (IY+n)

Instruktionerna är:

1. BIT b,x
   Testa om bit nummer b i cell x är en etta, om den är det nollställ Z-flaggan, om inte ettställ den.

2. RES b,x
   Nollställ bit nummer b i cell x.

3. SET b,x
   Ettställ bit nummer b i cell x.

9.5.6 PROGRAM FLOW CONTROL

Z80 erbjuder två sorters hopp: villkorliga och ovillkorliga.
För att hoppa ovillkorligt kan man använda:

1. JP x
   Hoppa till den position som anges av x, medger hopp till godtycklig adress.
   x är en av: nn = absolut adress, (IX) eller (IY).

2. JR e
   Hoppa till den position man får om man adderar PC och e. Detta kallas relativ adressering. E är en byte lång vilket ger en räckvidd på -128 till +127 byte. Den här instruktionen är kortare än JP och användes alltså med fördel överallt där dess räckvidd är tillräcklig.

CALL och RET motsvarar BASIC tolkens GOSUB och RETURN och användes alltså för att anropa subrutiner.
Om man i en subrutin vill använda registren bör man i allmänhet spara deras innehåll först. Detta göres lämpligen med PUSH

instruktionen. Innan RETuren måste man då POPa exakt lika många byte som man PUSHat, annars kommer RET att läsa en felaktig returadress från stacken.

3. CALL nn
   Spara nuvarande programräknare på stacken ("PUSH PC") och hoppa till nn.

4. RET
   Hämta en adress från stacken och hoppa till den position den anger ("POP PC").

CONDITION

			Un-cond.	Carry	Non carry	Zero	Non zero	Parity even	Parity odd	Sign neg	Sign pos	Reg B≠0
JUMP 'JP'	Immed. ext.	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	Relative	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
JUMP 'JP'	Reg. indir.	(HL)	E9									
JUMP 'JP'		(IX)	DD E9									
JUMP 'JP'		(IY)	FD E9									
'CALL'	Immed. ext.	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
Decrement B, jump if non zero 'DJNZ'	Relative	PC+e										10 e-2
Return 'RET'	Register indir.	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
Return from int 'RETI'	Reg. indir.	(SP) (SP+1)	ED 4D									
Return from non maskable int 'RETN'	Reg. indir.	(SP) (SP+1)	ED 45									

NOTE–CERTAIN FLAGS HAVE MORE THAN ONE PURPOSE. REFER TO SECTION 6.0 FOR DETAILS

JUMP, CALL and RETURN GROUP

Fig 9.5

All avbrottshantering måste avslutas med en av instruktionerna RETI eller RETN. Vilken som ska användas är man tvungen att hålla reda på själv.

5. RETI
   Retur från ett maskbart avbrott.

6. RETN
   Retur från ett icke maskbart avbrott.

7. RST n
   Den här instruktionen är ett en bytes CALL till en av åtta fasta adresser.
   n (adressen) är en av 0, 8, 16, 24, 32, 40, 48 eller 56.

   ABC80 innehåller fasta rutiner som anropas med RST, se ROM tipsen.

För att kunna skriva meningsfulla program måste man ha möjlighet att välja bland olika alternativa vägar i programmet, detta görs med olika sorters villkorliga hoppinstruktioner.
De fyra första villkorliga hoppen kollar först om villkoret är sant, om det är det utföres samma hopp som i det ovillkorliga fallet, om inte fortsätter exekveringen i nästa instruktion.
c1 är en av: C, NC, Z, NZ, PE, PO, M eller P.
c2 är en av: C, NC, Z eller NZ.

1. JP c1, x

2. JR c2, e

3. CALL c1, x

4. RET c1

5. DJNZ e
   Det här är en instruktion som är speciellt lämpad för loopkontroll. Den dekrementerar register B och gör ett hopp om B inte blev noll. Om man vill genomlöpa en loop ett visst antal gånger laddar man alltså B med önskat värde och avslutar loopen med en DJNZ till loopens början.
   DJNZ använder relativ adressering (som JR).

Här följer en sammanfattning över de olika hoppinstruktionerna.

9.5.7 BLOCK MOVE AND SEARCH

Z80 har ett antal kraftfulla blockoperationer som på ett enkelt

sätt medger att man manipulerar data i ett område av minnet. LDDx och CPDx instruktionerna arbetar "baklänges" (mot lägre adresser) i minnet, detta kan vara bra om man t ex ska flytta block som överlappar.
Alla LDx instruktionerna använder tre 16bit register: HL, DE och BC. HL användes för att peka på källblocket, DE pekar på destinationen och BC anger hur många byte som ska flyttas. Alla dessa register ändras av LDx instruktioner.
Instruktionerna är:

1. LDI
   Flytta en byte, räkna upp HL och DE, räkna ner BC ett steg.

2. LDIR
   Som LDI men den repeteras tills BC blir noll.

3. LDD
   Flytta en byte, räkna ner HL, DE och BC ett steg.

4. LDDR
   Som LDD men den repeteras tills BC blir noll.

CPx instruktionerna använder HL och BC registren på samma sätt som LDx. Jämförelsen sker mot ackumulatorn och resultatet hamnar i F registret.

5. CPI
   Jämför en byte, räkna upp HL, räkna ner BC ett steg.

6. CPIR
   Som CPI fast den repeteras tills BC blir noll eller en byte som är lika med ackumulatorns värde påträffas.

7. CPD
   Jämför en byte, räkna ner HL och BC ett steg.

8. CPDR
   Som CPD fast repetetiv.

9.5.8 INPUT/OUTPUT

För att kommunicera med yttervärlden behöver man någon sorts in och ut matning, detta görs i Z80 med speciella I/O instruktioner. Dessa instruktioner finns i repeterande varianter som liknar LDx

och CPx instruktionerna. För att kunna använda dem krävs emellertid speciell hårdvara.
Grundinstruktionerna är:

1. IN A, (n)
   Läser en byte från enhet n och lägger den i ackumulatorn.

2. IN r, (C)
   Läser en byte från den enhet vars nummer står i register C till register r.
   r är en av A, B, C, D, E, H eller L.

3. OUT (n), A
   Skriver ackumulatorns innehåll till enhet n.

4. OUT (C), r
   Skriver innehållet i register r till den enhet vars nummer finns i register C. Samma register som för IN r, (C).

9.5.9 CPU CONTROL

Det finns ett antal instruktioner som manipulerar CPUns allmänna beteende, de är:

1. NOP
   No operation, gör ingenting. Användes när man måste fylla en cell med något ofarligt.

2. HALT
   Stoppar processorn tills ett avbrott uppstår, eller maskinen RESETas. I ABC80 sker ett avbrott per 20 millisekunder från realtidsklockan.

3. DI
   Disable interrupt, stänger av avbrottssystemet.
   Observera att det finns två avbrottslinjer in till processorn, den ena (NMI) har man ingen möjlighet att stänga av.

4. EI
   Enable interrupt, sätter på avbrottssystemet.

5. IMO
   Sätter Z80 i avbrottsmod 0. Motsvarar 8080´s avbrott.

6. IM1
   Avbrottsmod 1, avbrott orsakar ett CALL till 0038H.

7. IM2
   Avbrottsmod 2, avbrott orsakar ett CALL via den adress som anges av dels register I (MSB) och dels av den enhet som orsakade avbrottet (LSB). Detta är alltså ett sant vektoriserat avbrott.

9.6 Handassemblering

Innan man kör ett assembler program måste det översättas. Det vanligaste är att man har ett program (assemblator) som sköter den detaljen, om man inte har ett sådant måste man handassemblera.

Detta innebär att man med hjälp av en lista över de aktuella instruktionernas värde räknar ut hur programmets instruktioner ska se ut i binärt format. Det besvärliga kommer när man ska räkna ut adresser som refererar till en position som man ännu inte hunnit till. För att kunna veta vilken adress en instruktion längre fram i programmet har måste man också veta hur många instruktioner det finns fram till den och hur lång var och en är.

Detta får man i allmänhet inte reda på utan att göra hela assembleringen fram till instruktionen i fråga. Man måste alltså vid alla framåtreferenser vänta med att lägga ut ett addressvärde ett tag för att senare gå tillbaka till den aktuella instruktionen och komplettera den.

Andra besvärligheter dyker upp när man ska beräkna relativa adresser, speciellt mot lägre adresser.
Oundgängliga hjälpmedel när man använder assemblerprogram (oavsett hur man översätter dem) är de handböcker som ZILOG med flera ger ut. De innehåller exakta definitioner av vad de olika instruktionerna gör, vilka flaggor som påverkas osv. Se litteraturhänvisningarna.

9.6.1 HEXPEEK OCH HEXPOKE

Det här är ett par små BASIC program som är nyttiga att ha när man ska lägga ut och kontrollera kod i minnet.

Med HEXPEEK kan vi titta på minnet, och med HEXPOKE kan vi ändra det.

```
1000 REM * HEXPEEK
1010 REM * Titta på minnet
1020 REM
1030 DIM H$=16,I$=10
1040 H$="0123456789ABCDEF"
1050 DEFFNH1(I)=((I/16) AND 15)+1
1060 DEFFNH2(I)=(I AND 15)+1
1070 REM
1080 ONERRORGOTO 1080
1090 PRINT "Adress"; : INPUT A
1100 ONERRORGOTO 0
1110 H=INT(A/256) : GOSUB 1230
1120 H=A AND 255 : GOSUB 1230
1130 ; ´ ´;
1140 FOR I=A TO A+7
1150 H=PEEK(I)
1160 GOSUB 1230 : ; ´ ´;
1170 NEXT I
1180 ;
1190 ; "Mer?"; : GET A$
1200 ; CHR$(13);
1210 IF A$=CHR$(13) THEN 1080
1220 A=A+8 : GOTO 1110
1230 REM - subrutin för att skriva
1240 REM - en byte som hextal
1250 REM
1260 ; MID$(H$,FNH1(H),1);
1270 ; MID$(H$,FNH2(H),1);
1280 RETURN
```

Exempel 9.1 HEXPEEK

```
1000 REM * HEXPOKE
1010 REM * Skriv i minnet
1020 REM
1030 H$="0123456789ABCDEF"
1040 REM
1050 ONERRORGOTO 1050
1060 PRINT "Adress"; : INPUT A
1070 ONERRORGOTO 0
1080 PRINT A,
1090 INPUT I$
```

```
1100 IF LEN(I$)=2 THEN 1120
1110 PRINT "???" : GOTO 1080
1120 GOSUB 1210
1130 IF E9<>0 THEN 1050
1140 POKE A,H
1150 A=A+1
1160 GOTO 1080
1170 REM - subrutin som omvandlar
1180 REM - en hexadecimal sträng I$
1190 REM - till talet h
1200 REM
1210 H=0 : E9=0
1220 F1=INSTR(1,H$,MID$(I$,1,1))-1
1230 F2=INSTR(1,H$,MID$(I$,2,1))-1
1240 IF F1<0 OR F2<0 THEN 1270
1250 H=F1*16+F2
1260 GOTO 1280
1270 E9=99
1280 RETURN
```

Exempel 9.2 HEXPOKE

9.7 Editor och assemblator

Om man ska använda maskinkod i större omfattning måste man ha en assemblator. För att kunna använda sin assemblator måste man ha en texteditor till hjälp att hantera programtexterna. Båda dessa program finns lyckligtvis tillgängliga för ABC80.

Här följer en genomgång av några grundläggande begrepp i en assemblator.

1. Symboler
   En symbol är ett namn som man har satt ett värde på. Detta värde kan vara av många olika typer och en symbols egenskaper bestämms nästan helt av värdets typ.

2. Lägen
   När man utför hopp sker dessa nästan uteslutande till förutbestämda positioner. För det mesta förser man dessa positioner med en symbol. Denna symbol kallas för ett läge.

3. Opkoder
   Även dessa är symboler, de användes för att ange vilken operation som ska läggas ut.

4. Direktiv
   För att styra assemblatorns beteende använder man direktiv. Saker som kan göras är att t ex reservera utrymme för variabler och konstanter och/eller definiera olika sorters symboler.

Här följer en lista med de direktiv som är tillåtna i ABC80:s assembler.

1. ORG nn
   Definierar vart i minnet ett program ska placeras. Detta måste bestämmas innan man kan assemblera och ladda ett program, om man vill flytta på det måste man assemblera om.

2. EQU nn
   Användes för att definiera en konstant. För att det här direktivet ska vara meningsfullt måste det finnas ett "läge" först på raden. Detta "läge" får nn som värde i stället för positionsräknarens värde.

3. END
   Ett program måste avslutas med END direktivet till skillnad mot BASIC program där det kan utelämnas.

4. DEFB n eller DEFB "c"
   Reserverar en cell (byte) i minnet och lägger dit antingen värdet av n eller ASCIIvärdet av c.

5. DEFW nn
   Reserverar två celler (ett ord, word) och lägger värdet av nn där.

6. DEFS nn
   Reserverar ett nn byte långt datablock i minnet, areans innehåll är odefinierat.

7. DEFM "ccc"
   Reserverar ett utrymme där strängen "ccc" precis får plats. Strängen får innehålla max 255 tecken.

9.8 Kommunikation mellan BASIC och assembler

En vanlig användning av assemblerrutiner är att ersätta vissa stycken (som måste exekvera fort) i ett BASIC program med maskinkod. För att göra det på ett effektivt sätt måste man ha möjlighet att skicka parametrar mellan delarna. Det finns i princip tre olika sätt att göra detta:

1. Via Z80 register
   BASIC funktionen CALL gör att man kan anropa en assemblerrutin med ett argument som hamnar i register DE. När rutinen gör RET användes register HL som funktionens resultat.
   Format: LET Z = CALL(adress, argument) Argumentet (och kommat) kan utelämnas.

2. Via fasta minnesceller
   Man lägger här upp några celler i sin assemblerrutin som man låter BASIC tolken använda genom att göra PEEK och POKE. Aningen lättare än metod 3 men den kräver att BASICprogrammet vet exakt var dessa celler ligger i minnet, detta ställer till elände om man ändrar något i sin assemblerrutin så att cellernas läge förändras.

3. Via BASIC variabler
   BASIC tolken håller alla användarens variabler i en lista (symboltabellen) som man kan komma åt från sina assemblerprogram. Varje variabel finns där komplett med namn, typ och värde. För att använda detta för kommunikation utser man en BASICvariabel (t ex Q9) till kanal och låter BASIC tolken använda den på vanligt sätt samtidigt som assemblerrutinen letar reda på den i listan och manipulerar den på lämpligt sätt.

9.9 Exempel — Drivrutin JOYSTICK

Som ett exempel på hur assemblerkod användes kommer vi nu att visa hur en joystickdrivare i assemblerkod ser ut. Fördelarna med att ha den i assembler är att vi med bibehållen noggrannhet kan sampla med betydligt större hastighet än i BASIC versionen.

Anrop av denna rutin sker som:

```
100 X%=CALL(65408,1)
110 Y%=SWAP%(X%) AND 255%
120 X%=X% AND 255%
```

På rad 100 anropar vi maskinkodsrutinen på adress 65408 och skickar med argumentet "1" till den i register DE. X% får sedan som värde det resultat som rutinen skickar tillbaka i register HL. Värdet är i detta fall Y koordinat i den mest signifikanta byten och X koordinat i den minst signifikanta. Raderna 110 och 120 plockar isär detta resultat.

Först tittar vi på programlistningen:

```
ABC80-assembler PASS 1

LOC   OBJ KOD     KÄLL KOD
                  ;     Drivrutin Joystick
                  ;
                  ;     Användning:
                  ;     Vid inhopp: DE = 1 eller 2 (joystick 1/2)
                  ;     Vid retur : H = X-räknare (0-79)
                  ;                 L = Y-räknare (0-79)
                  ;     Programmet läggs i poke-arean
                  ;     ORG 65408
                  ;
                  OFFSET EQU -18      ;Noll-justering;
                  ;
FF80 7B           START LD A,E
FF81 E603             AND 03H
FF83 287B             JR Z,NOPAR      ;Felaktig parameter
FF85 5F               LD E,A          ;Sense-bit-mask
                  ;
FF86 1608             LD D,8          ;X-trigg
FF88 CD0000           CALL COUNT
                  ;
FF8B 4D               LD C,L
                  ;
FF8C 1610             LD D,16         ;Y-trigg
FF8E CD0000           CALL COUNT
                  ;
FF91 61               LD H,C
FF92 C9               RET             ;Åter till BASIC
                  ;
                  ;     Räknerutin
```

```
                ;     Trigga vald monovippa
                ;     och mät pulslängden
                ;
FF93 AF         COUNT XOR A
FF94 D33A             OUT (58),A
FF96 7A               LD A,D
FF97 D33A             OUT (58),A    ;Trigga
FF99 060A             LD B,10
FF9B 1063       WAIT  DJNZ WAIT     ;Vänta lite
FF9D 210000           LD HL,OFFSET
                ;
                ;     Loop som mäter pulslängd
                ;
FFA0 23         PULSE INC HL
FFA1 DB3A             IN A,(58)     ;Läs porten
FFA3 A3               AND E         ;Sense-bit
FFA4 285A             JR Z,PULSE    ;Igen om puls kvar
                ;
                ;     Justering av värdet
                ;     Värden <0 eller >79 blir 0 resp. 79
                ;
FFA6 CB24       RANGE SLA H         ;Negativt ?
FFA8 3056             JR NC,RAOR1
FFAA 2E00             LD L,0        ;Om HL<0
FFAC 1852             JR RAEND
FFAE 3E4F       RAOR1 LD A,79
FFB0 BD               CP L
FFB1 304D             JR NC,RAEND
FFB3 6F               LD L,A        ;Om HL>79
FFB4 C9         RAEND RET
                ;
                ;     Feluthopp
                ;
FFB5 D7         NOPAR RST 16        ;Fel antal eller
FFB6 8D               DEFB 13+128   ;typ av argument
                      END
```

Fig 9.6 Drivrutin i assembler.

Beskrivning av rutinen:

1. Initiering
   Koden börjar med några kommentarer samt två direktiv. Direktiven definierar dels var i minnet programmet ska läggas, dels att symbolen "OFFSET" ska ha värdet -18.

2. Kontroll av parametern
   De tre första instruktionerna kollar om parametern är korrekt (anger vilken av två möjliga joysticks det gäller). Om den inte är det anropas en felrutin som ligger sist i koden.

3. Huvudprogram
   Efter att ha laddat register E med en konstant som behövs senare kommer programmets huvudmodul.
   Denna laddar en bitmask (8 för X, 16 för Y) som anger vilken bit på utporten som ska användas. För att spara kod anropas sedan subrutinen COUNT, sparandet sker genom att den anropas två gånger, dels för X dels för Y. Efter att ha laddat resultatet i HL (obs att det första resultatet sparades med instruktionen LD C,L) sker sedan returen.

4. Subrutinen COUNT
   Det är här som det egentliga arbetet utförs. Vi börjar med att nollställa A registret (XOR A!!) för att sedan skriva detta till port nummer 58. För att sedan generera en flank på porten skriver vi också den medskickade parametern (register D).
   Sedan väntar vi med hjälp av DJNZ instruktionen i ca 45 mikrosekunder på att joysticken ska reagera.
   Efter att ha laddat HL som användes som loopräknare, börjar vi räkna. För varje varv gör vi följande: inkrementera räknaren, kontrollera vad som finns på inporten, maska fram den bit som gäller för aktuell joystick och hoppa till början om det inte var klart.
   När vi räknat klart kontrollerar vi vad vi fått in. Om det var negativt sätter vi noll som resultat i stället och returnerar till huvudprogrammet. Om det var större än 79 sätter vi det till 79 och returnerar. Denna klippning gör att eventuella fel i hårdvaran aldrig orsakar att rutinen lämnar helt felaktiga resultat.

   När talet är kontrollerat och ev. justerat returnerar vi.

5. Felhantering
Om vi detekterar ett fel i inparametern hoppar vi till rutinen NOPAR som i sin tur hoppar till BASIC tolkens felhanterare. Detta gör att vi får en felutskrift av typen ERR 13.
Dessutom avbryts programmet man kör på ett snyggt sätt.

9.10 Tips på subrutiner i BASIC-interpretatorn

BASIC-ROMet innehåller en stor mängd subrutiner man kan använda i sina assemblerprogram. Tyvärr är det bara en del av dessa som har en fast och oföränderlig startadress, flertalet kan komma att flytta på sig i och med nya versioner av ABC80.
Här följer en uppräkning av några kända rutiner med garanterat fast startadress.
(Notation: nummer adress namn)

1. 0000H RESET
Ett hopp hit har samma verkan som att trycka på RESETknappen.

2. 0002H INCHAR
Läser ett tecken från tangentbordet och lägger det i register A. Tecknet ekas inte (man måste alltså skriva ut det själv om det ska synas på skärmen). Innehållet i DE och IX ändras.

3. 0005H INLINE
Läser en rad (avslutad med RETURN) från tangentbordet. Vid anropet måste HL peka på den minnesarea där resultatet ska hamna och BC innehålla det maximala antalet tecken som får matas in. Hela raden ekas.

4. 000BH OUTSTR
Skriver text på skärmen där markören råkar stå. Vid anrop ska HL peka på texten som ska skrivas och BC innehålla dess längd.
En finess är att man kan positionera markören på godtyckligt ställe innan utskriften sker om man börjar strängen med tecknen <ESC> och "=". <ESC> är ASCII tecken nummer 27. Dessa två tecken följs av två bytes med koordinater+32, ordning: X,Y.

5. 0010H ERROR1
Får BASIC tolken att ta kontrollen och skriva ut ett felmeddelande.
Anrop:
RST 10H
DEFB 128 + felnummer

6. 0011H ERROR2
Samma som ERROR1 fast anropet ser ut som:
LD A,128 + felnummer
JP 11H

7. 0044H LOOKUP
En rutin som söker efter en text i en tabell. Tabellen består av texter separerade med bytes som har bit7=1 och slutar med en byte=255. Resultatet från en lyckad sökning är värdet av separatorbyten före den matchande texten i tabellen, denna byte är då lämpligen 128 + index i en tabell med adresser. Vid anrop ska DE peka på tabellens början (som bara får innehålla stora bokstäver) och HL på texten man letar efter. En lyckad sökning markeras genom att rutinen kommer tillbaka med F registrets Zbit=1, A innehåller då separatorbyten.

Här följer dessutom några "flytande" rutiner. För att kunna kontrollera om en aktuell ABC80 har dem på de här angivna adresserna ger vi även de tre första byten i varje rutin. Om de inte finns på exakt rätt ställe hittar man dem troligen nära den gamla positionen.
(Notation: namn nummer adress byte1 byte2 byte3)

1. CLEAR 0276H 3EH 18H DDH
Tömmer skärmen, förstör innehållet i IX, DE och A.

2. CURXY 0293H E5H 2AH F3H
Beräknar adressen i bildminnet till position X, Y på skärmen. Resultatet kommer i DE.
Innan man anropar denna rutin måste man lägga koordinaterna i därför avsedda celler: X i 65011H, Y i 65012H.
X betecknar position i höjdled och Y i sidled.

3. FNDLIN 0F39H 2AH 1CH FEH
Letar reda på början av den BASIC rad vars nummer finns i DE. Resultatet är en adress i HL.

4. ASC2I 181EH 7EH 11H 00H
   Översätter en sträng med siffror till ett binärt tal. HL ska peka på strängen vid anropet, resultat i DE. Om carry är satt vid returen gick det fel av någon anledning. Strängen anses sluta i och med första tecken som inte är en siffra.

5. I2ASC 1862H AFH 01H F0H
   Översätter ett binärt tal till en sträng. HL innehåller talet, BC en pekare till en strängbuffert.

6. IMUL 3774H 42H 4BH 11H
   Heltalsmultiplikation: (HL)*(DE) => HL, carry=fel uppstod.

7. IDIV 3788H 7AHB3H 37H
   Heltalsdivision: (HL)/(DE) => HL, carry=fel uppstod.

KAPITEL 10

Anslutning av egna enheter

ABC80

10.1 Inledning

Hur stor eller liten dator man än har, kommer alltid det ögonblicket då man vill koppla in något mer till den. Det kan vara mera minne, eller en liten anordning som sätter på kaffepannan klockan sju på morgonen, reglerar trafikljusen i Mjölby eller något annat.

Till ABC80 finns en mängd olika utbyggnadskort att köpa - D/A-omvandlare, A/D-omvandlare, minneskort, kort med reläutgångar osv osv.

I det här kapitlet ska vi titta lite närmare på hur vi ansluter en enhet till ABC80, framför allt hur den programmässiga anslutningen går till.

Men först måste vi bestämma oss för den fysiska anslutningen!

10.2 Fysisk anslutning

De utbyggnadskort som finns att köpa är gjorda för att anslutas till I/O-bussen, utom minneskorten som av naturliga skäl är gjorda för minnesbussen (I/O-bussen och minnesbussen finns bägge tillgängliga i busskontakten på ABC80´s baksida, men principerna för anslutning till de två bussarna skiljer sig markant).

Men, när vi själva vill bygga ihop något, finns det i princip tre olika vägar att gå. Vilken vi väljer beror dels på hur många olika signaler vi behöver sända till och från enheten, dels på hur pass komplicerad vi vill göra den.

De tre metoderna är:

1. Anslutning via V24-kontakten.
   Det här är den minst komplicerade metoden, men den har samtidigt minst möjligheter. Vi kan bara ansluta en enhet i taget till V24-kontakten, och antalet signaler är bara 3 in och två ut. Detta kan dock räcka många gånger, speciellt om det är seriell överföring det gäller, och JOYSTICK´en är ett exempel på en enhet som är ansluten den här vägen.

   Om vår enhet inte drar för mycket ström, kan den ta +12 V och -12 V från V24-utgången, så slipper vi ett speciellt nätaggregat för den.

2. I/O-bussen.
   Det här är den naturliga anslutningen av utbyggnadskort. Upp till 64 stycken kan adresseras av ABC80. Konstruktionen blir lite mera komplicerad än vid V24-alternativet, eftersom kortet måste innehålla s k CARD-SELECT logik, dvs endast ETT kort i taget får vara utvalt att "prata" med ABC80. Dessutom finns det några kontrollsignaler att hålla reda på (t ex signalen som kallas IN7 ska innebära "RESET" till alla kort på I/O-bussen).

   Totalt kan 8 databitar och 9 kontrollsignaler överföras.

3. Minnesbussen.
   Det här är den naturliga platsen för extra minneskort, men vi kan även tänka oss att bygga upp andra enheter så, att ABC80 ser dem som en (eller flera) minnesceller. I stället för OUT portnummer,data / INP(portnummer), refererar vi enheten med POKE adress, data / PEEK(adress).

   En fördel med denna metod, "minnesmappad I/O", är att vi inte först behöver göra CARD-SELECT, utan enheten finns hela tiden färdig att kommunicera med.

   Nackdelarna är främst två:

   1. Kortet blir det mest komplicerade av våra tre alternativ eftersom vi måste göra en fullständig avkodning av alla 16 adressbitarna. Detta är tvunget för att inte adresskollisioner ska kunna uppstå (dvs kortets adresser är de samma som något annat korts, eller som någon del av ABC80´s existerande minne).

   2. VAR (på vilka adresser) kan vi lägga enheten? Hela ABC80´s minnesrymd är ju (eller är snart) förutbestämd för ett eller annat ändamål!

      Den "säkraste" platsen - dvs de platser som kommer att vara lediga längst - är mellan adresserna 29696 och 30719. Det är 1 Kbyte som ligger mellan IEC-ROMmet och printerROMmet, och som kallas "ledigt" på minneskartan i ABC80´s bruksanvisning. Observera att IEEE-kortet också tar detta minnesutrymme i anspråk.

      Det allra bästa är förstås att enhetens adress kan väljas med strömbrytare direkt på kortet, men då blir det ju ännu dyrare och ännu mer komplicerat!

Slutsatsen av det här resonemanget blir:

Ska vi bygga egna enheter, så ska vi helst göra dem anpassade för I/O-bussen. Då passar de garanterat på vilken ABC80 som helst.

Vi går här inte in närmare på hur enhetsanpassningen går till rent elektriskt. Den som vill veta mer kan läsa i (ref. 10).

Vi ska i stället gå över till den programmässiga anslutningen.

10.3 Programmässig anslutning

Nu har vi virat eller lött ihop vår enhet (ev köpt den). Hur ska vi kommunicera med den?

Ja, det beror dels vilken typ av enhet det är, dels på hur pass mycket programmeringsjobb vi kan tänka oss att lägga ner.

Precis som allting annat, så sönderfaller det här problemet i tre delar:

1. Vi kan "prata" med enheten direkt i BASIC, som vi gjorde med JOYSTICK´en i kap. 7. Dvs vi gör INP och OUT (ev PEEK och POKE) för att kommunicera. Detta går an, om vi inte har alltför bråttom - uppemot 300 st INP eller OUT i sekunden går faktiskt att göra!

2. Vi skriver en drivrutin i maskinkod, och gör CALL på den. Jämför vår behandling av JOYSTICK´en i kapitel 9.

3. Vi definierar enheten som en fil ("logisk enhet"). Detta kräver att vi skriver en drivrutin i maskinkod OCH att vi länkar in enhetsnamnet och rutinadressen i enhetslistan. Exempel på användning:

```
OPEN "JOY:" ASFILE 1
.
.
INPUT #1;X%,Y%
```

Ur den rene användarens synpunkt är förstås den tredje metoden trevlig. Han behöver inte alls tänka på hur kommunikationen med

enheten fungerar, det är bara att skriva PRINT och INPUT i programmet.

Men, för oss som ska skriva drivrutinen blir det naturligtvis mera jobb. Dessutom måste ju rutinen länkas in i ABC80´s enhetslista. Varje gång vi slår av spänningen eller gör "RESET", så försvinner all information om eventuella extra enheter som vi har länkat in, och länkningen måste alltså göras om. Detta ordnar vi enklast med ett speciellt BASIC-program för varje enhet som ska länkas. Detta program stoppar dels in maskinkoden i minnet, och dels länkar det in enheten i enhetslistan.

Den som sedan ska använda sig av enheten måste komma ihåg att efter spänningstillslag och "RESET" köra detta program (Exempel RUN SKAPAJOY, eller något liknande).

Eftersom metod 1 och 2 ovan enbart beror på enheten, lämnar vi dem härmed, och koncentrerar hela vår skapelse på metod 3 (så svårt är det ändå inte?).

10.4 Att definiera en enhet som fil

10.4.1 HOPPTABELL, ENHETSLÄNK, PARAMETERBLOCK

En drivrutin för en logisk enhet ska alltid börja på ett bestämt sätt, nämligen med en hopptabell, där följande hopp ska finnas (nödvändigtvis i nämnd ordning):

```
JPTAB   JP      OPEN    ; öppna enheten för läsning
        JP      PREPARE ; skapa en fil på enheten
        JP      CLOSE   ; stäng enheten
        JP      INPUT   ; läs in en rad från enheten
        JP      PRINT   ; skriv på enheten
        JP      BL_IN   ; läs ett block ( 256 byte ) från
                        ; enheten
        JP      BL_UT   ; skriv ett block
```

Dessutom måste vi ha en länk till enhetslistan:

```
LÄNK    DEFW    0       ; plats för gamla enhetspekaren
```

```
DEFM    "NAM"    ; enhetens namn ( 3 bokstäver )
DEFW    JPTAB    ; adressen till hopptabellen
```

Vid inlänkning av enheten tar vi värdet av enhetspekaren (cellerna 65034 - 65035) och lägger i de två byten vid LÄNK. Sedan tar vi adressen till LÄNK och lägger i 65034 och 65035.

När en fil öppnas, skapar BASIC-tolken ett "parameterblock", som tillhör filen. Detta block består av 15 byte, och adressen till blocket finns i indexregister IX när drivrutinen anropas (det ser tolken till). Varje fil som är öppnad har ett eget block, så en och samma drivrutin kan utan problem användas till flera filer på en gång.

I parameterblocket ligger lite information (för tolken och för drivrutinen), men det finns även lite plats för lokala variabler, om drivrutinen skulle behöva det. De olika bytarna i blocket är definierade som följer:

IX+0,IX+1	Pekare till nästa öppna fil, 0 om inga fler. (information för tolken)
IX+2	Logiskt filnummer. Satt till N vid OPEN "NAM:" ASFILE N.
IX+3,IX+4	Pekare till enhetens post i enhetslistan. (information för tolken)
IX+5	0 om filen är stängd, 1 om den är öppen.
IX+6	Innehåller den position där nästa tecken skall skrivas ut. (Lokal variabel för drivrutinen).
IX+7	Innehåller max antal positioner på en rad. (Lokal för drivrutinen).
IX+8	Reserverad för tolken.
IX+9 - IX+14	Fritt fram för lokala värden.

Det finns även en alternativ betydelse av bytarna IX+8 - IX+14, som gäller när vi använder oss av blockning av data. Blockning innebär att data som skrivs, samlas upp i en buffert om 256 byte, och först när den är full sker det egentliga utskrivandet på enheten genom anrop av drivrutinens BLOCK_UT - delrutin (och motsvarande vid inläsning).

Betydelsen av byte IX+8 - IX+14 blir då:

IX+8,IX+9	Adress till buffert.
IX+10,IX+11	Aktuell adress i bufferten (att skriva i eller läsa från).
IX+12	Fritt fram (den enda helt lediga cellen).
IX+13	Innehåller antal lediga byte i bufferten.
IX+14	Status: Bit 7 = 0 => inget skrivet i bufferten = 1 => data skrivna i bufferten Bit 0 = 0 => aktuell buffert ej utskriven = 1 => aktuell buffert utskriven

Se vidare om blockning vid beskrivning av de olika delrutinerna.

De olika delrutinerna i drivrutinen måste göra vissa bestämda saker, och måste givetvis veta var de kan hitta olika parametrar - filnamn t ex Här följer en sammanfattning av de olika delrutinernas uppgifter och parametrar.

Observera att rutinerna hela tiden beskrivs som om det vore någon form av skrivare de styrde (det pratas om "raden full" etc.). Naturligtvis behöver det inte alls vara så, men ideerna i rutinerna är sådana att det tydligast framgår vad de SKA göra, om man betraktar enheten som en skrivare. När det gäller andra typer av enheter, t ex ett D/A-omvandlarkort behöver vi givetvis inte hålla reda på "position på raden" o dyl utan gör i stället det som är relevant för det kortet.

Observera också att rutinerna inte får förändra värdet på register IX och IY.

Om vi ändå vill använda IX och IY, kan rutinerna börja med att göra PUSH och avsluta med att göra POP på dessa register.

```
        PUSH    IX              ; spara register
        PUSH    IY
        .                       ; här är rutinen
        .
        .
        POP     IY              ; återställ register
        POP     IX
        RET                     ; återvänd
```

10.4.2 OPEN, PREPARE

Denna delrutin anropas av BASIC-tolken vid OPEN "NAM:" ASFILE 1, PREPARE "NAM:" ASFILE 1, LIST NAM:, LOAD NAM: och RUN NAM:.

Givetvis kan OPEN och PREPARE ha varsin rutin, men de är väldigt lika, så här behandlar vi dem på samma sätt.

Vid inträdet i rutinen är IX satt att peka på parameterblocket, och register DE pekar på (det eventuella) filnamnet som angetts. Observera att punkten i filnamnet är borttagen av tolken.

Här måste vi göra vissa initieringar, t ex lägga värdet för maximal radlängd i (IX+7). Om vår enhet inte är något skrivarliknande utan kanske ett kort med D/A-omvandlare, kan vi lämpligtvis använda (IX+7) att lägga kortadressen i (den adress som skall anges vid CARD-SELECT).

Exempel:

```
OPEN    .                           ; gör vad som skall göras
        .
        .
        LD      (IX+7),40           ; max radlängd
        LD      (IX+6),0            ; aktuell position
        AND     A                   ; nollställ CARRY
        RET                         ; och återvänd
```

Om vi skall använda oss av BLOCKNING är det ännu mer som måste göras i OPEN. Då blir det:

```
OPEN    .
        .
        LD      HL,BUFF             ; adressen till en buffert
        LD      (IX+7),40           ; radlängd (t.ex)
        LD      (IX+8),L            ; buffertadressen
        LD      (IX+9),H            ; kom ihåg att 16-bitars tal
                                    ; ligger swappade
        LD      (IX+10),L           ; första lediga adress i buf.
        LD      (IX+11),H           ;
        LD      (IX+13),253         ; 253 byte kvar i bufferten
        LD      (IX+14),0           ; inget skrivet någonstans
        LD      (HL),3              ; lägg ETX i bufferten
        .
        .
        .                           ; etc.
```


Om OPEN (PREPARE) av någon anledning inte lyckas, kan vi tala om det för BASIC-tolken genom att anropa felrutinen:

```
        RST     10H                 ; anropa felrutin
        DEFB    felnummer+128       ; ge felnummer
```

10.4.3 CLOSE

Denna rutin anropas vid CLOSE 1. Här är inte så mycket att göra:

```
CLOSE   .                           ; gör vad du ska
        .
        .
        AND     A                   ; nollställ CARRY
        RET                         ; återvänd
```

Lite mer blir det om vi använder blockning. Då måste vi testa bit 7 i (IX+14) för att se om det finns data i bufferten som inte är ivägsända till filen, och om så är fallet får vi ta hand om de data som är kvar.

10.4.4 INPUT

Anropas vid INPUT #1, INPUTLINE #1, LOAD NAM: och RUN NAM: . Register HL innehåller adressen där texten ska läggas, och BC innehåller maximalt antal tecken som får läsas in.

Observera att det skall vara en textsträng, avslutad med CR (CHR$(13)), mycket viktigt, som ska läggas i (HL) även om det i BASIC-programmet står INPUT #1,A% ! Detta medför, om enheten t.ex är ett A/D-omvandlarkort som levererar data i en byte, att vi själva måste omvandla dessa binära data till en textsträng, som i sin tur återomvandlas till binära data av BASIC-tolken!

```
INPUT   LD      A,C
        AND     A                   ; jämför med 0
        JP      Z,FULLT
        .
        <hämta ett tecken från enheten till A-reg.>
        .
        LD      (HL),A              ; lagra det i strängen
        INC     HL
```

```
        DEC     C
        CP      ODH             ; var det RETURN ?
        JP      NZ,INPUT        ; om inte, fortsätt med nästa
        JP      KLART           ; var det RETURN är vi klara
FULLT   DEC     HL
        LD      (HL),ODH        ; stoppa in ett RETURN sist
KLART   AND     A               ; nollställ CARRY
        RET                     ; återvänd
```

Om vi vill signalera "slut på filen" till tolken ska vi sätta CARRY innan vi hoppar tillbaka (och A-registret ska vara 0).

```
EOF     XOR     A               ; nollställ A-reg.
        SCF                     ; ettställ CARRY
        RET                     ; återvänd
```

Detta genererar fel nummer 34, som BASIC-progammet kan ta hand om.

Om vi använder oss av blockning blir input-rutinen mycket kort (Se 10.4.6 BLOCK_IN). Den finns nämligen redan färdigskriven åt oss, i BASIC-ROMmet, och blir bara:

```
INPUT   JP      0015H           ;
```

10.4.5 PRINT

Anropas vid PRINT #1 och LIST NAM: .

I register HL finns adressen till den sträng av tecken som ska skrivas ut, och BC innehåller antal tecken. Observera att även om vi i vårt BASIC-program gör PRINT #1,A% så får drivrutinen en STRÄNG att skriva, nämligen samma sträng som om vi hade gjort PRINT #1,NUM$(A%).

Rutinen måste hålla reda på om den utskrivna raden har blivit full (dvs om (IX+6) >= (IX+7)), och i så fall skriva ut en radframmatning och nollställa (IX+6). I annat fall skrivs bara tecknen ut, och (IX+6) ökas motsvarande.


```
PRINT   LD      A,C
        OR      B               ;testa om BC = 0
        RET     Z               ; återvänd i så fall
        LD      A,(HL)          ; hämta ett tecken från
                                ; strängen
        .
        <skriv ut ett tecken>
        .
        INC     HL
        DEC     BC
        INC     (IX+6)          ; öka position med ett
        LD      A,(IX+6)
        CP      (IX+7)          ; jämför med maxpos.
        JR      NZ,PRINT        ; OK tag nästa
        .                       ; raden full !
        <behandla full rad>
        .
        JP      PRINT           ; och fortsätt
```

När vi använder blockning blir print-rutinen lika enkel som i input-fallet (Se 10.4.7 BLOCK_UT). Även denna rutin finns färdig.

```
PRINT   JP      001BH           ; hoppa till ROMmet, som
                                ; självt gör return !
```

10.4.6 BLOCK_IN

Denna rutin anropas automatiskt av input-rutinen (0015H) när bufferten är helt färdigläst och vad vi ska göra är "bara":

```
BL_IN   <läs in ett block från enheten till bufferten>
        LD      HL,BUF          ; adressen till bufferten
        RET
```

10.4.7 BLOCK_UT

Denna rutin är nästan likadan:

```
BL_UT   <skriv ut ett block på enheten>
        LD      (IX+13),253     ; 253 byte lediga igen
        SET     0,(IX+14)       ; markera att bufferten är
                                ; utskriven
```

```
LD      HL,BUF          ; addressen till bufferten
EX      DE,HL
RET
```

10.5 Konventioner

Här, precis som på alla andra områden, är det fördelaktigt att ha några konventioner att följa.

För I/O-kort, som alltså är anslutna via I/O - bussen, är det föreslaget att det ska se ut så här:

```
OPEN "NAM:Sxxx" ASFILE 1
```

Där "NAM" står för enhetens namn, "S" är kortets adress (CARD-SELECT nummer) och "xxx" står för andra eventuella parametrar vi vill sända med till rutinen.

Vi bör ena oss om en viss standard för vilka addresser som ska användas till vad. I appendix-I finns ett sådant förslag. Se även Ref-11.

Exempel ett A/D - kort med kortadress 17:

```
OPEN "AD:17" ASFILE 1
.
.
PRINT #1,11 : REM Kanalval
INPUT #1,K(11) : REM Läs ett värde
.
.
CLOSE 1
```

För enheter anslutna till V24-kontakten (eller minnesmappade) antar vi samma standard, men utan kortvalsnummer.

Exempel en skrivare ansluten via V24-kontakten:

```
OPEN "V24:PA.3" ASFILE 1
```

Där "P" står för pariteten, "A" för hur många extra tecken skrivaren behöver efter CR och LF, och trean står för hastigheten 1200 baud.

Om vi har ett printerinterface på I/O - bussen i stället kan det bli:

```
OPEN "PR:1PA.3" ASFILE 1
```

där 1 alltså är kortvalsnummer, och bokstäverna har samma betydelse som tidigare.

KAPITEL 11

Datakommunikation

ABC80

11.1 Inledning – några begrepp

Datakommunikation är nog det område inom datatekniken som utvecklas snabbast just nu. Terminaler av olika slag som står i förbindelse med centrala datorsystem blir allt vanligare.

Hur kan man då som ABC80-användare ha nytta av datakommunikation?

Jo, det kan gälla allt, ifrån att sända över ett nytt BASIC-program till en god vän i Korpilombolo, till att använda ABC80 som en registreringsterminal till en stordator.
Det kan också bli frågan om att söka information i en Datavisionscentral.

Som ett kuriosum kan nämnas att texten du just nu läser har knappats in på en ABC80 som stått i dialog med en större dator. T80PRT är ett program som gör ABC80 till en terminal varvid det som skrivs på tangentbordet sänds iväg på linjen och mottagna data från linjen skrivs på skärmen. På direktiv från ABC80 har sedan den större datorn producerat ett offset-underlag som gått till tryckeriet.

Den kanal som överföringen sker på kan vara av olika slag.

Fig 11.1 ABC80 kopplad till akustiskt modem.

Om inte avståndet är för stort kan vi lägga ut en egen kabel. Vi kan också använda oss av telefonnätet och ringa upp den vi önskar kontakta eller vi kan hyra en linje av televerket. Eftersom telefonnätet är gjort för tal (analoga signaler) krävs en anpassningsenhet i form av ett MODEM (MOdulator/DEModulator) som omvandlar våra digitala "ettor" och "nollor" till tonfrekvenser och omvänt.

Vi kan också ansluta oss till televerket nya allmänna datanät som är digitalt. Mera om detta längre fram.

Gemensamt för de kanaler som nämts ovan är att vi vanligen bara har tillgång till ett trådpar och vi måste därför överföra data seriellt.

Datasignaleringshastigheten mäts i bitar per sekund, BPS, som vid binär signalering är lika med modulationshastigheten som mäts i BAUD.
En uppringd telefonlinje klarar som regel inte mera än 1200 baud med ett vanligt modem. Ett akustiskt modem klarar 300 baud.

Om sändning bara kan ske i en riktning talar vi om SIMPLEX-överföring. Med DUPLEX betecknar man en förbindelse där sändning kan ske i båda riktnigar samtidigt. Kan sändning ske i båda riktningar, fast ej samtidigt, heter det HALV DUPLEX. Genom att använda flera frekvenser kan man arbeta i duplex även på bara ett trådpar.

Behovet av standard gör sig speciellt gällande vid datakommunikation. Att båda utrustningarna använder samma teckenkod är ju väsentligt t ex Vi har tidigare träffat på ASCII-kod (Appendix-B) som ett exempel på standard. (ASCII = American Standard for Information Interchange) . Det finns dock flera andra kodrepresentationer, t ex EBCDIC som bl.a. IBM-datorer använder.
Det finns flera organisationer som utarbetar standards och rekommendationer. En av dessa är CCITT (Comite Consultatif International de Telegraphique et Telephonique) . ISO (International Standard Organisation) är en annan.

11.2 Elektriskt gränssnitt — V.24

Vi har tidigare använt V24-kontaktens in- och ut-gångar utan att bekymra oss om att den följer en viss standard. (Kap 7) . Hur ledarna används vid datakommunikation ska vi nu titta närmare på.

Den fullständiga CCITT-rekommendationen specificerar hela 50 ledare men av dessa används vanligen bara dessa:

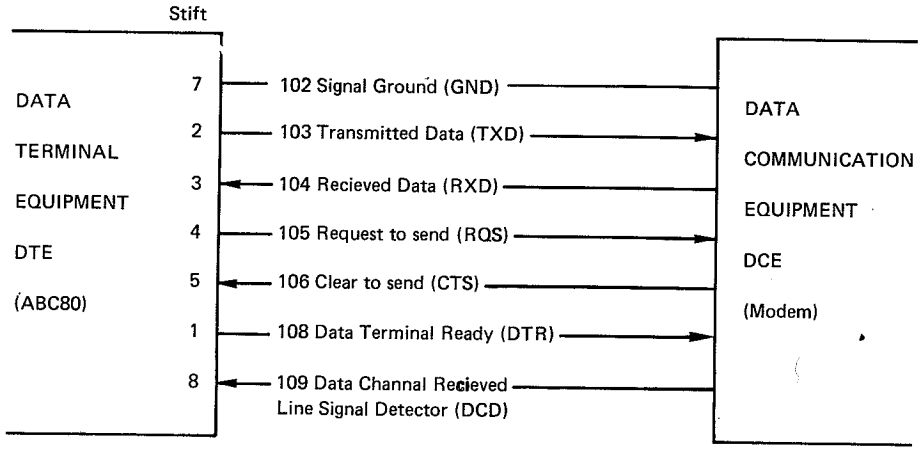


Fig 11.2 V24-snittet.

En negativ spänning betyder "1" (-3 till -25 V) .

En positiv spänning betyder "0" (+3 till +25 V) .

En annan beteckning på gränssnitt man brukar träffa på är RS232C vilket är nästan samma sak som V.24 enligt ovan.

11.3 Asynkron överföring — ett exempel

Seriell överföring kan ske på två sätt, synkront eller asynkront.

Vid synkron överföring är tidsavståndet mellan de enskilda bitarna i ett tecken konstant och avståndet mellan två tecken är alltid ett helt antal bitintervall.

Vid asynkron överföring är avståndet mellan två tecken däremot godtyckligt.

Problemet är i båda fallen att mottagaren måste veta när varje bit kommer och när varje tecken börjar. Vid synkron överföring sänds speciella synkroniseringstecken (SYN) för att mottagaren ska kunna "hålla takten".

I det asynkrona fallet måste varje tecken innehålla information som gör att mottagaren kan känna igen var tecknet börjar. För detta går det åt extra bitar och den asynkrona överföringen blir därmed långsammare. Varje tecken består av tre delar: Ett startelement, databitarna och ett eller flera stoppelement. (CCITT rekommendation V.3)

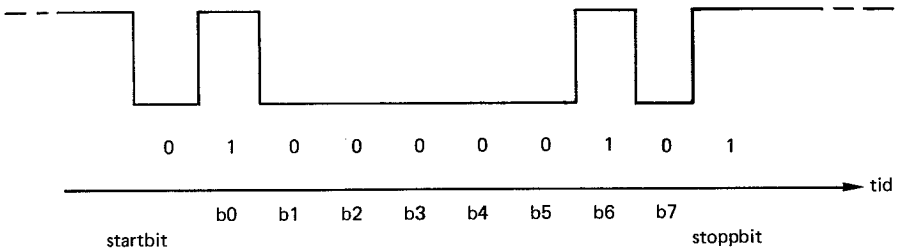


Fig 11.3 Överföring av bokstaven "A" (41H = 010000001B).

Linjen är vid vila alltid "1". Varje tecken börjar med en nolla som startbit. Efter databitarna kommer en eller flera stoppbitar som sätter linjen till "1" igen.

Som vi ser går det här åt 10 bitar för varje tecken. Det betyder att med en hastighet av 1200 baud kommer maximalt 120 tecken per sekund att kunna överföras.

Nu kan det ju vara så att mottagaren inte kan ta emot tecknen i den takt som sändaren förmår sända, t ex därför att text ska skrivas ut på en långsam skrivare. För att kunna "hejda" sändaren används signalerna RTS och CTS.

Omvandlingen av tecknen till seriellt format kan ske i hårdvara (SIO) eller i en maskinspråksrutin. Vi kan faktiskt även göra det i BASIC - upp till 300 baud!

```
10 GET A$ : D%=ASC(A$)
20 GOSUB 1000 : REM * Sändtkn *
30 GOTO 10
.
.
.
880 REM ----------------------------
890 REM ¦ Sändtkn
900 REM ----------------------------
910 REM Tecknet CHR$(D%) sänds ut på
920 REM V24-kontakten i 300 baud.
930 REM Fås ej CTS inom 10 sekunder
940 REM sker felindikering i E9%.
950 REM
960 REM Invariabel :
970 REM D%  Tecken som ska sändas
980 REM E9% Ev. felkod
990 REM ----------------------------
1000 REM * Sändtkn *
1010 E9%=0%
1020 REM * Bitvektor-seq *
1030 D%(1%)=0% OR 16% : REM Startbit
1040 FOR I%=2% TO 9%
1050 D%(I%)=((D% AND 1%)*8%) OR 16%
1060 D%=D%/2%
1070 NEXT I%
1080 D%(10%)=8% OR 16% : REM Stoppbit
1090 REM * Bitvektor-end *
1100 REM * RTS-seq *
1110 OUT 58%,8% OR 16%
1120 FOR I%=0% TO 10000%
1130 IF (INP(58%) AND 2%)=0% THEN 1170
1140 NEXT I%
1150 E9%=42% : REM Enheten ej klar
1160 GOTO 1250
1170 REM * RTS-end *
1180 REM * TXD-seq *
1190 FOR I%=1% TO 10%
1200 OUT 58%,D%(I%)
1210 FOR I2%=0% TO 7% : NEXT I2%
1220 NEXT I%
1230 REM * TXD-end *
1240 REM * Sändtkn-end *
1250 RETURN
```

Exempel 11.1 SÄND300

Vi har "trimmat" sändaren till 300 baud m.h.a. fördröjningsloopen på rad 1210.

Denna subrutin kan användas för att sända ut text till en skrivare som har V24-snitt eller till en annan ABC80, eventuellt via modem.

Vill vi koppla ihop två ABC80 med en egen kabel (max 15-20 m) kan vi göra så här:

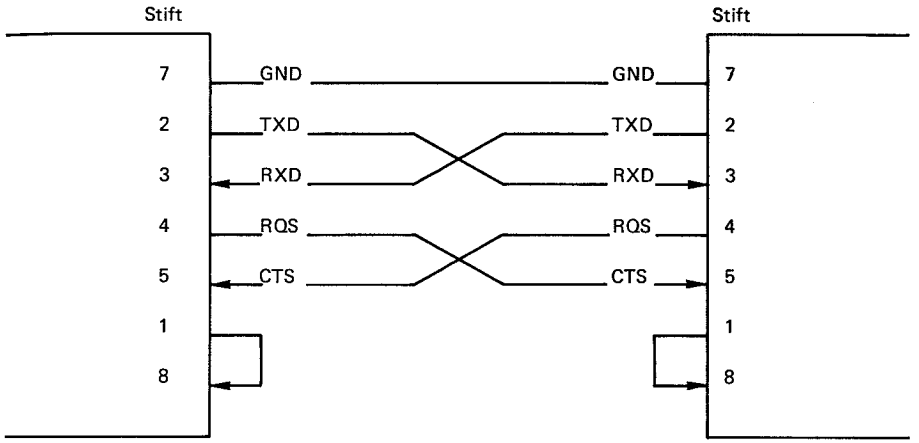


Fig 11.4

Då behöver vi förstås även en mottagningsrutin:

```
10 GOSUB 2000 : REM * Mottagtkn *
20 ; CHR$(D%);
30 GOTO 10
.
.
.
1910 REM ----------------------------
1920 REM ¦ Mottagtkn
1930 REM ----------------------------
1940 REM Ett tecken CHR$(D%) mottas
1950 REM från V24-kontakten i 300 baud.
1960 REM Utvariabel :
1970 REM D%  Mottaget tecken
1980 REM E9% Ev. felkod
1990 REM ----------------------------
2000 REM * Mottagtkn *
```

```
2010 E9%=0%
2020 REM * CTS-seq *
2030 OUT 58%,8%
2040 IF (INP(58%) AND 2%)=0% THEN 2040
2050 REM * CTS-end *
2060 REM * RXD-seq *
2070 IF (INP(58%) AND 1%)=1% THEN 2070
2080 FOR I%=1% TO 10%
2090 D%(I%)=INP(58%) AND 1%
2100 FOR I1%=0% TO 7% : NEXT I1%
2110 NEXT I%
2120 REM * RXD-end *
2130 OUT 58%,8% OR 16%
2140 IF D%(1%)=0% AND D%(10%)=1% THEN 2160
2150 E9%=58% : REM Ogiltigt tecken
2160 REM * Tkn-seq *
2170 D%=0%
2180 FOR I%=9% TO 2% STEP -1%
2190 D%=D%*2%
2200 D%=D% OR D%(I%)
2210 NEXT I%
2220 REM * Tkn-end
2230 REM * Mottagtkn-end *
2240 RETURN
```

Exempel 11.2 MOTT300

Vi kan, genom att utnyttja dessa subrutiner, mycket väl sända över ett BASIC-program till vår vän i Korpilombolo om vi båda har modemer. Vi får då ringa upp vår vän och be honom ladda mottagningsprogrammet. Vi sänder över, frågar om det gick bra och lägger på luren.

Det var här frågan om överföring i bara en riktning. Om vi istället ringer upp en obemannad datacentral krävs ju mycket mera. Datacentralen måste automatiskt kunna svara på uppringningen. I och med att kommunikationen är dubbelriktad måste det kunna fastställas när det är vår tur att sända och när det är den andra datorns tur. Om det uppstår fel bör omsändning ske etc.

Man måste alltså ha noggranna styrregler för kommunikationen.

11.4 Logiskt gränssnitt — linjeprocedur

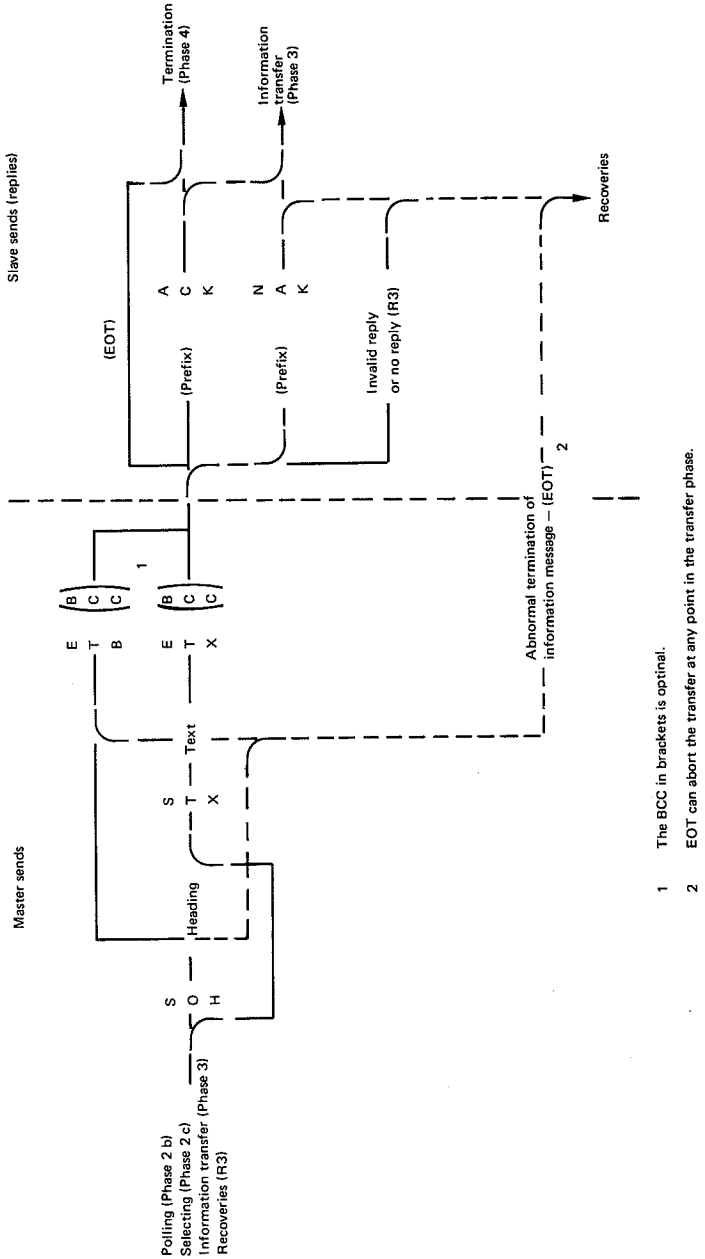


Fig 11.5 ISO 1745

De styrregler som behövs för att upprätta, vidmakthålla och bryta ned en förbindelse brukar man kalla en linjeprocedur eller ett protokoll. Tyvärr finns det ett otal olika protokoll som tagits fram av olika datortillverkare varför utrustningar av olika fabrikat inte alltid kan kopplas ihop utan vidare. Standardisering pågår dock.

Linjeprocedurer är ofta ganska komplicerade program på åtskilliga Kbyte maskinkod. Man kan skilja på två typer av procedurer, teckenorienterade och bitorienterade.

ISO-1745 är exempel på en teckenorienterad procedur. Ett datablock måste innehålla ett helt antal bytes. Överföringen styrs av kontrolltecken. (Jämför ASCII-tabellen, appendix-B) . Så inleds t ex ett datablock med tecknet SOH följt av ett huvud. Själva informationen följer efter tecknet STX och avslutas med ETX eller ETB.

Själva texten får inte innehålla något kontrolltecken. Varje block kan avslutas med en kontrollsumma BCC. Om överföringen av blocket gick bra kvitterar mottagaren med kontrolltecknet ACK. Vid NAK sker omsändning.

ISO-HDLC (High Level Data Link Control) är exempel på en bitorienterad procedur. Den är dessutom transparent vilket betyder att informationen kan bestå av en helt godtycklig bitföljd. För att avgränsa ett block används en flagga som består av 6 stycken ettor i följd.

2 FRAME STRUCTURE

In HDLC, all transmissions are in frames, and each frame conforms to the following format:

Flag	Address	Control	Information	FCS	Flag
01111110	8 bits	8 bits	*	16 bits	01111110

* An unspecified number of bits which in some cases may be a multiple of a particular character size, for example an octet.

where

Flag = flag sequence
Address = secondary station address field
Control = control field
Information = information field
FCS = frame checking sequence

Fig 11.6 Blockavgränsning i HDLC-proceduren.

Skulle själva informationen någonstans innehålla mera än 5 ettor i följd skjuts en extra nolla in av sändaren. En nolla efter 5 stycken ettor tas bort av mottagaren.

HDLC väntas få stor användning i framtiden.

IBM-2780 och PDP11-RXS är exempel på andra vanliga linjeprocedurer som finns till ABC80.

11.5 Det allmänna datanätet

Tills alldeles nyligen har dataöverföring varit hänvisad till nät som konstruerats för andra behov än data. Det gäller både telefonnätet och telexnätet.

Televerket har nu börjat ta i bruk ett digitalt nät som är speciellt uppbyggt för data. Datanätet är synkront och man kan abonnera på en hastighet från 600 till 9600 baud. Den anslutningsenhet som behövs för att ansluta sig till det allmänna datanätet heter DCE (Data Circuit-terminating Equipment) . Gränssnittet är X21 men utrustning med V24-snitt kan också anslutas med en särskild DCE som heter DCE-V. Även asynkron utrustning kan anslutas upp till en hastighet av 300 baud.

Datanätet erbjuder även ett antal tilläggstjänster som t ex direktanrop (alltid samma nummer) , spärr mot utgående samtal, vidarekoppling etc.

11.6 Datavision

Datavision, Viewdata eller Teledata är namnet på ett projekt som televerket driver, tills vidare på försök. Det är ett informationssökningssystem där man t ex med en ABC80, ett speciellt modem och en telefon kan koppla upp en datavisionscentral.

Exempel på tänkbara tillämpningar är:

* Nyheter, väder
* Samhällsinformation
* Konsumentupplysning
* Tidtabeller, biljettbokning
* Utbildning
* Uppslagsverk
* Annonser
* Information inom företag
* Beräkningar, spel
* Utvärdering av meddelanden.

Informationen presenteras i form av text och bilder på en skärm med 24 rader a 40 tecken och 72*78 grafiska punkter.
Det låter ju bekant. ABC80 har utformats för att passa datavision.

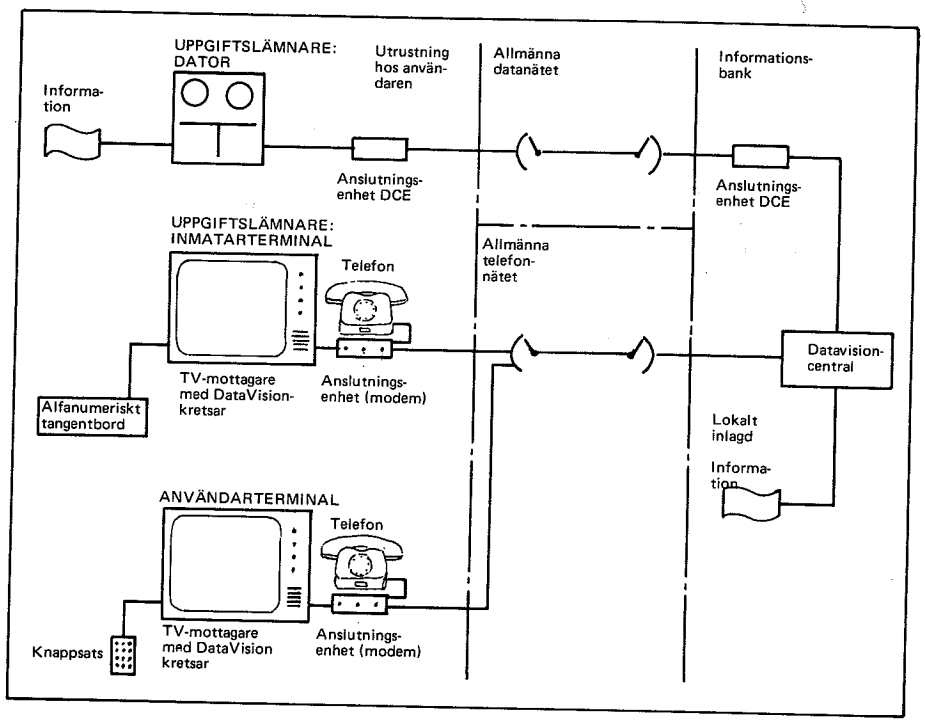


Fig 11.7 Principen för datavision.

Referenslitteratur

1. Bruksanvisning ABC80 / SCANDIA METRIC, LUXOR.

2. ABC om BASIC / Anders Andersson, Arne Kullbjer, Jan Lundgren, Sören Thornell / DIDACT.

3. Bruksanvisning FD2 och FD2U / SCANDIA METRIC, LUXOR.

4. Bruksanvisning ABC80 assembler / SCANDIA METRIC, LUXOR.

5. Z80-CPU Technical manual / ZILOG INC (SCANDIA METRIC).

6. Z80-PIO Technical manual / ZILOG INC (SCANDIA METRIC).

7. Programmering för mig och dig / Jan Lundgren, Bengt Lundin.

8. JSP - en praktisk metod för programkonstruktion / Leif Ingevaldsson / STUDENTLITTERATUR -77.

9. Hur datorer fungerar / Tord-Jöran Hallberg / STUDENTLITTERATUR -79.

10. Mikrodatorns ABC / Gunnar Markesjö / ESSELTE STUDIUM -78.

11. ABC om mätdatorsystem / Per Eriksson, Håkan Magnusson, Mats Rasmussson / SCANDIA METRIC.

Appendix

ABC80

Appendix A
Binärtal och hexadecimaltal

OMVANDLINGSTABELL

Decimalt	Binärt	Hexadecimalt
0	00000000B	00H
1	00000001B	01H
2	00000010B	02H
3	00000011B	03H
4	00000100B	04H
5	00000101B	05H
6	00000110B	06H
7	00000111B	07H
8	00001000B	08H
9	00001001B	09H
10	00001010B	0AH
11	00001011B	0BH
12	00001100B	0CH
13	00001101B	0DH
14	00001110B	0EH
15	00001111B	0FH
16	00010000B	10H
17	00010001B	11H
18	00010010B	12H
19	00010011B	13H
20	00010100B	14H
21	00010101B	15H
22	00010110B	16H
23	00010111B	17H
24	00011000B	18H
25	00011001B	19H
26	00011010B	1AH
27	00011011B	1BH
28	00011100B	1CH
29	00011101B	1DH
30	00011110B	1EH
31	00011111B	1FH
32	00100000B	20H

Appendix B
ASCII–tabell

SIS

SVENSK STANDARD **SIS 63 61 27**

SVERIGES STANDARDISERINGSKOMMISSION
STANDARDKOMMITTEN FÖR ADMINISTRATIV TEKNIK
OCH DATABEHANDLING

Utgåva 1 Sida 1 (7)
Första giltighetsdag 1976 - 07 - 01

Databehandling UDK 681.3.042

SVENSKA 7 BITARS TECKENKODER

Data processing. Swedish 7-bit coded character sets

Orientering

Denna standard ersätter SEN 85 02 00.

Standarden omfattar svenska teckenkoder med 7 bitar, dels en grundversion och dels en version som innehåller alla bokstäver som i officiella sammanhang används vid skrivning av svenska egennamn.

Som bilagor har för information medtagits den internationella referensversionen av teckenkod med 7 bitar samt en svensk teckenkod med 6 bitar.

I standarden angiven placering av tecknen Å, å, Ä, ä och Ö, ö har skett i samråd med standardiseringsorganen i Danmark, Finland, Norge och Västtyskland. Dessa länder och Sverige har lika bestämmelser för placeringen av tecknen Ä, ä och Ö, ö medan enligt tysk standard Ü och ü är placerade i de positioner där de nordiska länderna placerat Å och å.

I standarden kommer Å efter Ö. Denna placering har valts för att få internationell överensstämmelse och innebär inte någon rekommendation att ändra ordningsföljden i det svenska alfabetet. Vid programmering blir det i vissa fall nödvändigt att göra tillägg för att få konventionell svensk alfabetisk ordningsföljd.

Syftet med denna standard är att nå största möjliga enhetlighet vid informationsutbyte vid användning av olika datamedier.

Standarden är tillämpbar vid informationsutbyte mellan olika utrustningar för datakommunikation och på olika datamedier.

Hänvisning

ISO 646–1973, 7-bit coded character sets for information processing interchange

Med ISO 646 överensstämmande teckenkoder är:

CCITT (Comité Consultatif International de Télégraphie et Téléphonie) Alphabet No 5

ECMA (European Computer Manufacturers Association) – 6, 7-bit input/output coded character set, 1973

Nära överensstämmande är:

ASCII, USA standard ANSI X3.4–1968

1 Terminologi

Vissa av nedastående termer ingår i SEN 01 16 04, Databehandling, Ordlista, Organisering av data. Där används dock kvittenstecken, klocktecken osv.

1.1 Styrtecken

Beteckning	Benämning Engelsk	Svensk
ACK	Acknowledge	kvittens
BEL	Bell	klocka
BS	Backspace	backsteg (backtecken)
CAN	Cancel	ignorering
CR	Carriage Return	vagnretur
DC	Device Control	organstyrning
DEL	Delete	makulering
DLE	Data Link Escape	gruppskift
EM	End of Medium	mediumslut
ENQ	Enquiry	förfrågning
EOT	End of Transmission	överföringsslut
ESC	Escape	kodskift (skiftningstecken)
ETB	End of Transmission Block	blocköverföringsslut
ETX	End of Text	textslut
F	Function	funktion
FE	Format Effector	redigering
FF	Form Feed	blankettmatning
FS	File Separator	filgräns
GS	Group Separator	gruppgräns
HT	Horizontal Tabulation	horisontal tabulering
IS	Information Separator	gränstecken
LF	Line Feed	nedmatning
NAK	Negative Acknowledge	negativ kvittens
NL	New Line	radmatning*
NUL	Null	för utfyllnad (tomtecken)
RS	Record Separator	postgräns
SI	Shift-In	inskift
SO	Shift-Out	utskift
SOH	Start of Heading	rubrikinledning
SP	Space	blanksteg, mellanrum
STX	Start of Text	textinledning
SUB	Substitute	ersättning
SYN	Synchronous Idle	synkronisering
TC	Transmission Control	styrning vid dataöverföring
US	Unit Separator	elementgräns
VT	Vertical Tabulation	vertikal tabulering

*) Detta avser en sammankoppling av CR och LF varvid vanligen koden för LF används.

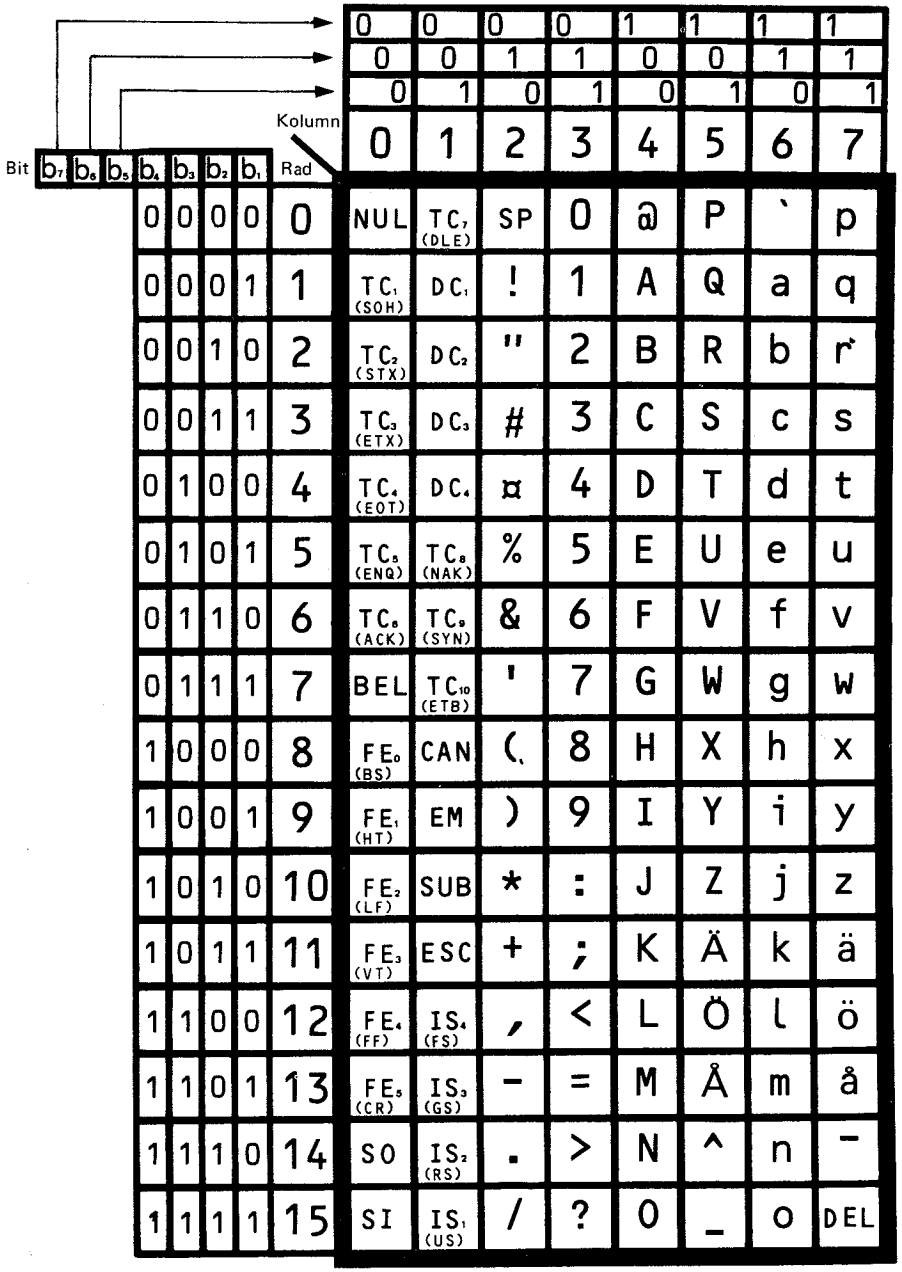
1.2 Grafiska hjälptecken

Position*	Symbol	Benämning
2/0		mellanrum
2/1	!	utropstecken
2/2	"	anföringstecken
2/3	#	nummertecken
2/4	¤	valutatecken
2/5	%	procenttecken
2/6	&	kommersiellt och
2/7	'	apostrof
2/8	(	vänsterparentes
2/9	)	högerparentes
2/10	*	asterisk
2/11	+	plus
2/12	,	komma
2/13	-	minus; bindestreck
2/14	.	punkt
2/15	/	snedstreck
3/10	:	kolon
3/11	;	semikolon
3/12	<	mindre än
3/13	=	likhetstecken
3/14	>	större än
3/15	?	frågetecken
4/0	@	kommersiellt å
5/11	[	vänsterhake
5/12	\	inverterat snedstreck
5/13	]	högerhake
5/14	^	cirkumflex
5/15	_	understreck
6/0	`	grav aksent
7/11	{	vänsterklammer
7/12	|	vertikalstreck
7/13	}	högerklammer
7/14	~	överstreck

*) Position anges efter mönstret kolumn/rad i tabellerna.

2 Svensk teckenkod med 7 bitar, Grundversion, avsedd för allmän användning

Teckenkoden har registrerats inom ISO och fått beteckningen **ESC 2/8 4/7.**

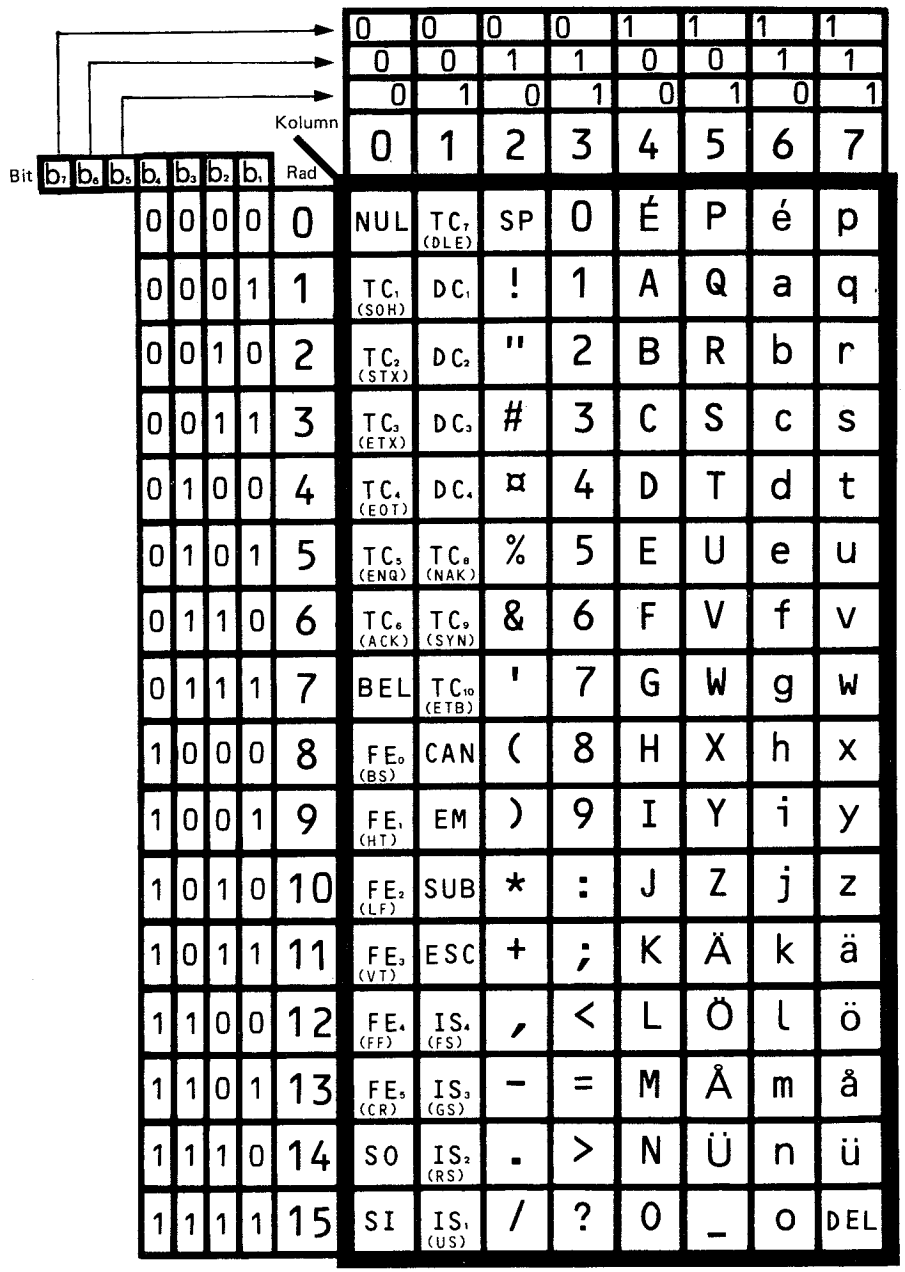


Bit b_7	b_6	b_5	b_4	b_3	b_2	b_1	Rad	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1
							Kolumn	0	1	2	3	4	5	6	7
0	0	0	0				0	NUL	TC_7 (DLE)	SP	0	@	P	`	p
0	0	0	1				1	TC_1 (SOH)	DC_1	!	1	A	Q	a	q
0	0	1	0				2	TC_2 (STX)	DC_2	"	2	B	R	b	r
0	0	1	1				3	TC_3 (ETX)	DC_3	#	3	C	S	c	s
0	1	0	0				4	TC_4 (EOT)	DC_4	¤	4	D	T	d	t
0	1	0	1				5	TC_5 (ENQ)	TC_8 (NAK)	%	5	E	U	e	u
0	1	1	0				6	TC_6 (ACK)	TC_9 (SYN)	&	6	F	V	f	v
0	1	1	1				7	BEL	TC_{10} (ETB)	'	7	G	W	g	w
1	0	0	0				8	FE_0 (BS)	CAN	(	8	H	X	h	x
1	0	0	1				9	FE_1 (HT)	EM	)	9	I	Y	i	y
1	0	1	0				10	FE_2 (LF)	SUB	*	:	J	Z	j	z
1	0	1	1				11	FE_3 (VT)	ESC	+	;	K	Ä	k	ä
1	1	0	0				12	FE_4 (FF)	IS_4 (FS)	,	<	L	Ö	l	ö
1	1	0	1				13	FE_5 (CR)	IS_3 (GS)	-	=	M	Å	m	å
1	1	1	0				14	SO	IS_2 (RS)	.	>	N	^	n	~
1	1	1	1				15	SI	IS_1 (US)	/	?	O	_	o	DEL

Exempel: Koden för bokstaven A är 1000001

3 Svensk teckenkod med 7 bitar, för skrivning av egennamn i officiella sammanhang

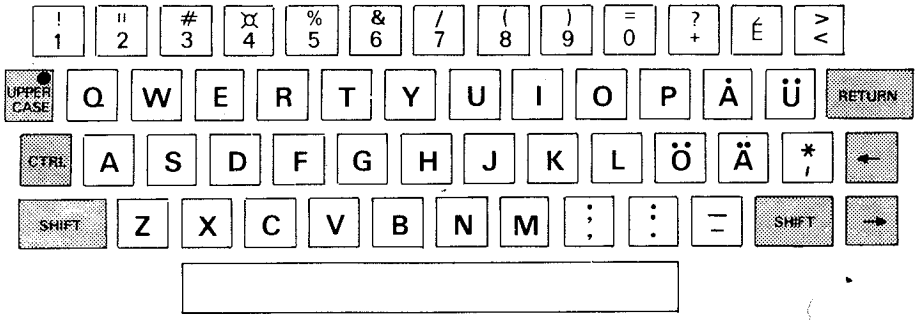
Teckenkoden har registrerats inom ISO och fått beteckningen **ESC 2/8 4/8.**



Bit b_7	b_6	b_5	b_4	b_3	b_2	b_1	Rad	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1
							Kolumn	0	1	2	3	4	5	6	7
0	0	0	0				0	NUL	TC_7 (DLE)	SP	0	É	P	é	p
0	0	0	1				1	TC_1 (SOH)	DC_1	!	1	A	Q	a	q
0	0	1	0				2	TC_2 (STX)	DC_2	"	2	B	R	b	r
0	0	1	1				3	TC_3 (ETX)	DC_3	#	3	C	S	c	s
0	1	0	0				4	TC_4 (EOT)	DC_4	¤	4	D	T	d	t
0	1	0	1				5	TC_5 (ENQ)	TC_8 (NAK)	%	5	E	U	e	u
0	1	1	0				6	TC_6 (ACK)	TC_9 (SYN)	&	6	F	V	f	v
0	1	1	1				7	BEL	TC_{10} (ETB)	'	7	G	W	g	w
1	0	0	0				8	FE_0 (BS)	CAN	(	8	H	X	h	x
1	0	0	1				9	FE_1 (HT)	EM	)	9	I	Y	i	y
1	0	1	0				10	FE_2 (LF)	SUB	*	:	J	Z	j	z
1	0	1	1				11	FE_3 (VT)	ESC	+	;	K	Ä	k	ä
1	1	0	0				12	FE_4 (FF)	IS_4 (FS)	,	<	L	Ö	l	ö
1	1	0	1				13	FE_5 (CR)	IS_3 (GS)	-	=	M	Å	m	å
1	1	1	0				14	SO	IS_2 (RS)	.	>	N	Ü	n	ü
1	1	1	1				15	SI	IS_1 (US)	/	?	O	_	o	DEL

Exempel: Koden för bokstaven A är 1000001

Appendix C
Tangentbordslayout och koder

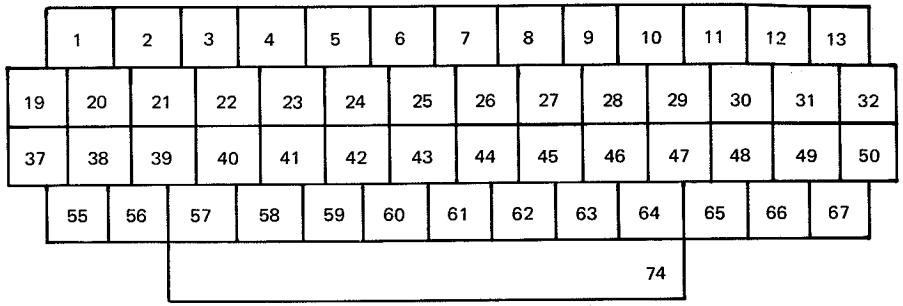


T = Tangent nummer
S = Med SHIFT
CS = Med CONTROL-SHIFT

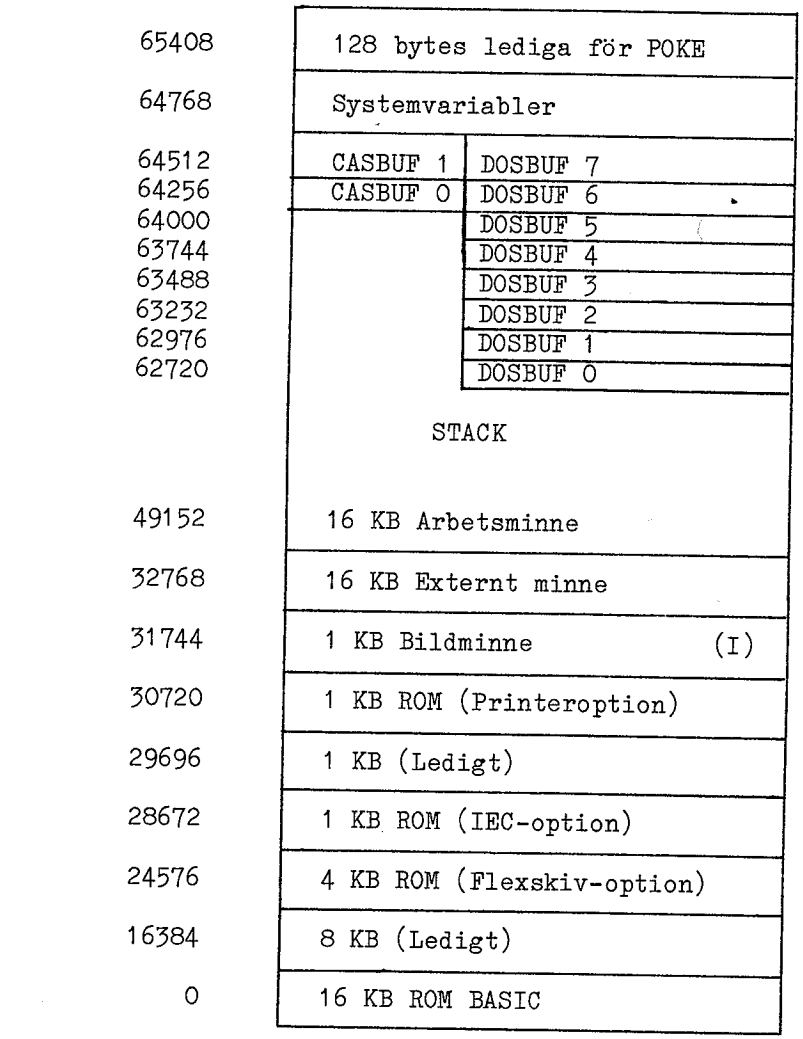
U = Utan SHIFT
C = Med CONTROL

(Oktalt, Hexadecimalt)

T	U	S	C	CS	T	U	S	C	CS
1	061 31	041 21	061 21	041 21	32	015 0D	015 0D	015 0D	015 0D
2	062 32	042 22	062 32	042 22	38	141 61	101 41	001 01	001 01
3	063 33	043 23	063 33	043 23	39	163 73	123 53	023 13	023 13
4	064 34	944 24	964 34	044 24	40	144 64	104 44	004 04	004 04
5	065 35	045 25	065 35	045 25	41	146 66	106 46	006 06	006 06
6	066 36	046 26	066 36	046 26	42	147 67	107 47	007 07	007 07
7	067 37	057 2F	067 37	057 2F	43	150 68	110 48	010 08	010 08
8	070 38	050 28	070 38	050 28	44	152 6A	112 4A	012 0A	012 0A
9	071 39	051 29	071 39	051 29	45	153 6B	113 4B	013 0B	033 1B
10	060 30	075 3D	060 30	075 3D	46	154 6C	114 4C	014 0C	034 1C
11	053 2B	077 3F	053 2B	077 3F	47	174 7C	134 5C	034 1C	034 1C
12	140 60	100 40	000 00	000 00	48	173 7B	133 5B	033 1B	033 1B
13	074 3C	076 3E	177 7F	177 7F	49	047 27	052 2A	047 27	052 2A
20	161 71	121 51	021 11	021 11	50	010 08	010 08	010 08	010 08
21	167 77	127 57	027 17	027 17	56	172 7A	132 5A	032 1A	032 1A
22	145 65	105 45	005 05	005 05	57	170 78	130 58	030 18	030 18
23	162 72	122 52	022 12	022 12	58	143 63	103 43	003 03	003 03
24	164 74	124 54	024 14	024 14	59	166 76	126 56	026 16	026 16
25	171 79	131 59	931 19	931 19	60	142 62	162 42	002 02	002 02
26	165 75	125 55	025 15	025 15	61	156 6E	116 4E	016 0E	036 1E
27	151 69	111 49	911 09	011 09	62	155 6D	115 4D	015 0D	035 1D
28	157 6F	117 4F	017 0F	037 1F	63	054 2C	072 3B	054 2C	073 3B
29	160 70	120 50	020 10	000 00	64	056 2E	072 3A	056 2E	072 3A
30	175 7D	135 5D	035 1D	035 1D	65	055 2D	137 5F	055 2D	137 5F
31	176 7E	136 5E	036 1E	036 1E	67	011 09	011 09	011 09	011 09



Appendix D
Minneskarta ABC 80



(I) Se vidare Appendix E

Appendix E
Bildskärm och grafik

RADERNAS ADRESSER I BILDMINNET

Rad	Decimal adress
0	31744 - 31783
1	31872 - 31911
2	32000 - 32039
3	32128 - 32167
4	32256 - 32295
5	32384 - 32423
6	32512 - 32551
7	32640 - 32679
8	31784 - 31823
9	31912 - 31951
10	32040 - 32079
11	32168 - 32207
12	32296 - 32335
13	32424 - 32463
14	32552 - 32591
15	32680 - 32719
16	31824 - 31863
17	31952 - 31991
18	32080 - 32119
19	32208 - 32247
20	32336 - 32375
21	32464 - 32503
22	32592 - 32631
23	32720 - 32759

Dessa radadresser finns lagrade i BASIC-ROMet från adress 884 till 911.

Kod	T	G	Kod	T	G	Kod	T	G	Kod	T	G
32	Blank		56	8		80	P	P	104	h	
33	!		57	9		81	Q	Q	105	i	
34	"		58	:		82	R	R	106	j	
35	#		59	;		83	S	S	107	k	
36	¤		60	<		84	T	T	108	l	
37	%		61	=		85	U	U	109	m	
38	&		62	>		86	V	V	110	n	
39	'		63	?		87	W	W	111	o	
40	(		64	É	É	88	X	X	112	p	
41	)		65	A	A	89	Y	Y	113	q	
42	*		66	B	B	90	Z	Z	114	r	
43	+		67	C	C	91	Ä	Ä	115	s	
44	,		68	D	D	92	Ö	Ö	116	t	
45	-		69	E	E	93	Å	Å	117	u	
46	.		70	F	F	94	Ü	Ü	118	v	
47	/		71	G	G	95	–	–	119	w	
48	0		72	H	H	96	é		120	x	
49	1		73	I	I	97	a		121	y	
50	2		74	J	J	98	b		122	z	
51	3		75	K	K	99	c		123	ä	
52	4		76	L	L	100	d		124	ö	
53	5		77	M	M	101	e		125	å	
54	6		78	N	N	102	f		126	ü	
55	7		79	O	O	103	g		127	■	

Koder tolkade i teckenmod (T) och grafmod (G)

Appendix F
Programmet PROCENT

Program : PROCENT
Version : V1
Utrustn.: ABC 80, FD2
Förf. : Örjan Kärrsgård / Comporian AB
Datum : 79-08-30

Innehåll:

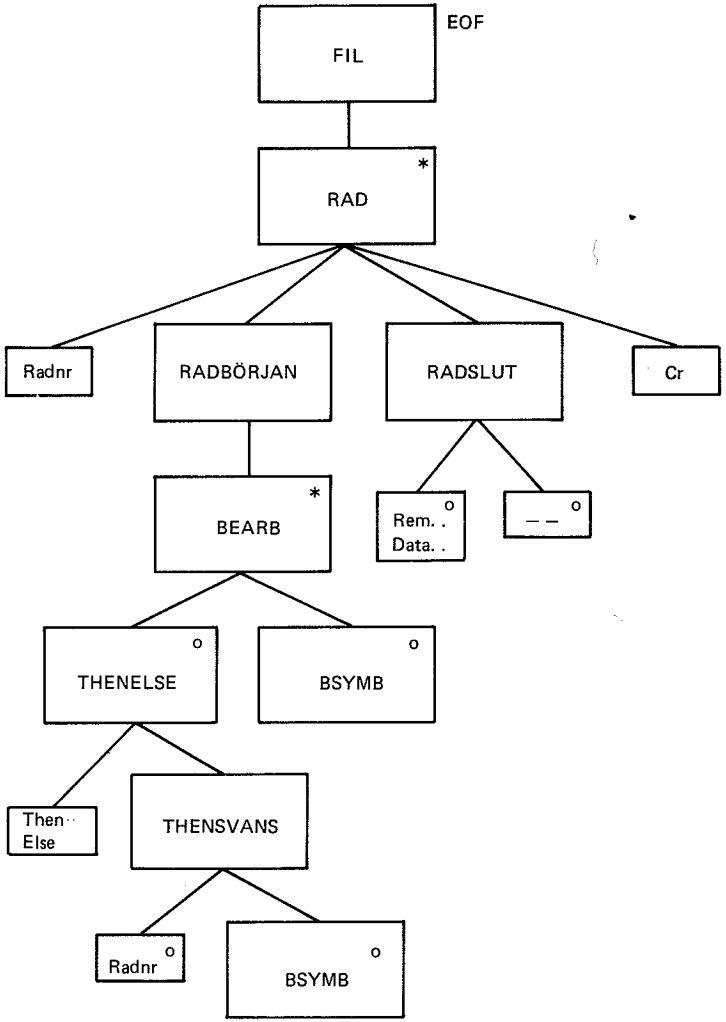
1. FUNKTIONSBESKRIVNING

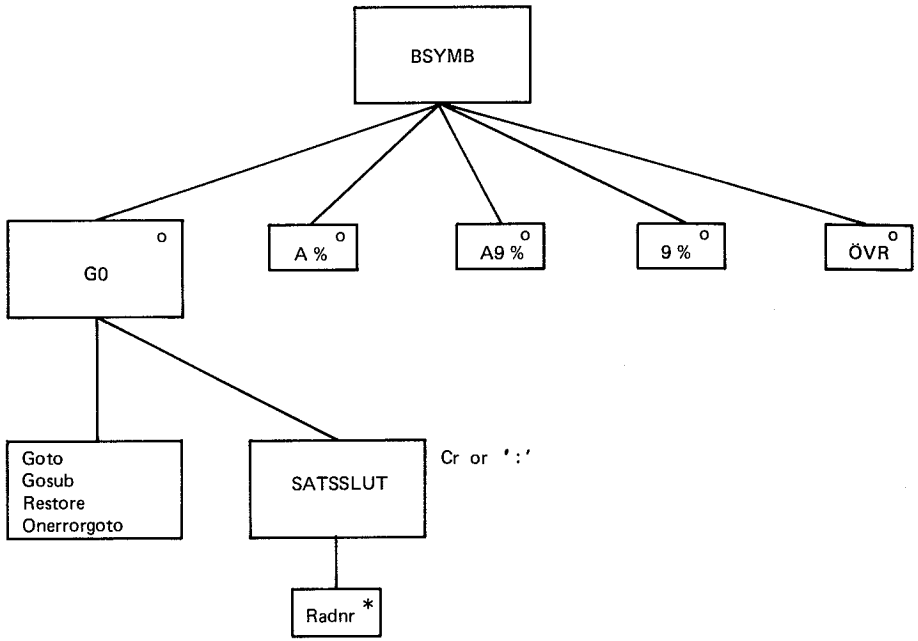
Programmet PROCENT är ett hjälpmedel vid programutveckling. Programmet ändrar alla variabler och konstanter som är flyttal till heltal genom att lägga till %-tecken på dessa. Program som enbart arbetar med heltal blir därigenom snabbare och tar mindre plats i minnet.

Indata till PROCENT är en programfil på diskett lagrad som text (.BAS). Som resultat fås en ny programfil.

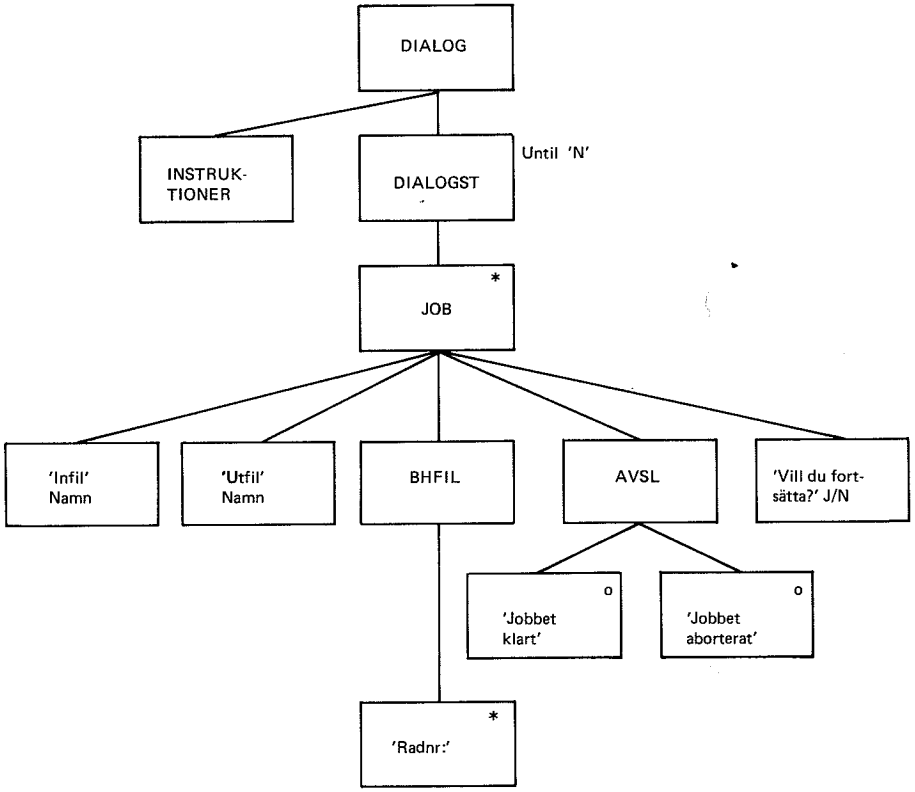
2. DATASTRUKTURER OCH VARIABELBESKRIVNING.

2.1 Datastruktur in- och utfil





2.2 Datastruktur dialog

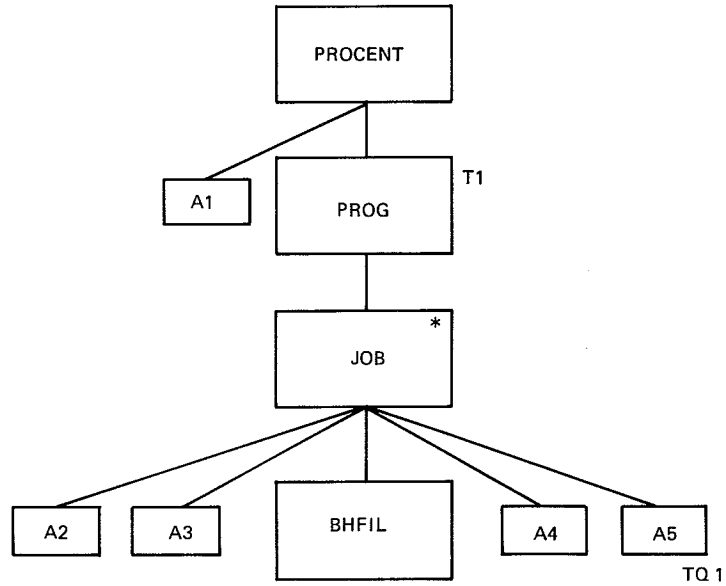


2.3 Variabelbeskrivning

Dedicerade variabler:

D1$	Rad från infil. Den del som ej behandlats.
D2$	Rad till utfil. Den del som behandlats.
D	Ascii-kod för senast inlästa tecken.
B$	Aktuell symbol.
B	ASC(B$) eller Cr (RETURN).
E9	Felkod.

3. PROGRAMSTRUKTUR MED OPERATIONSLISTA



Termineringsvillkor:

T1,TQ1 Ej "J" eller "j" på frågan "Vill du fortsätta" i A5.

Operationer:

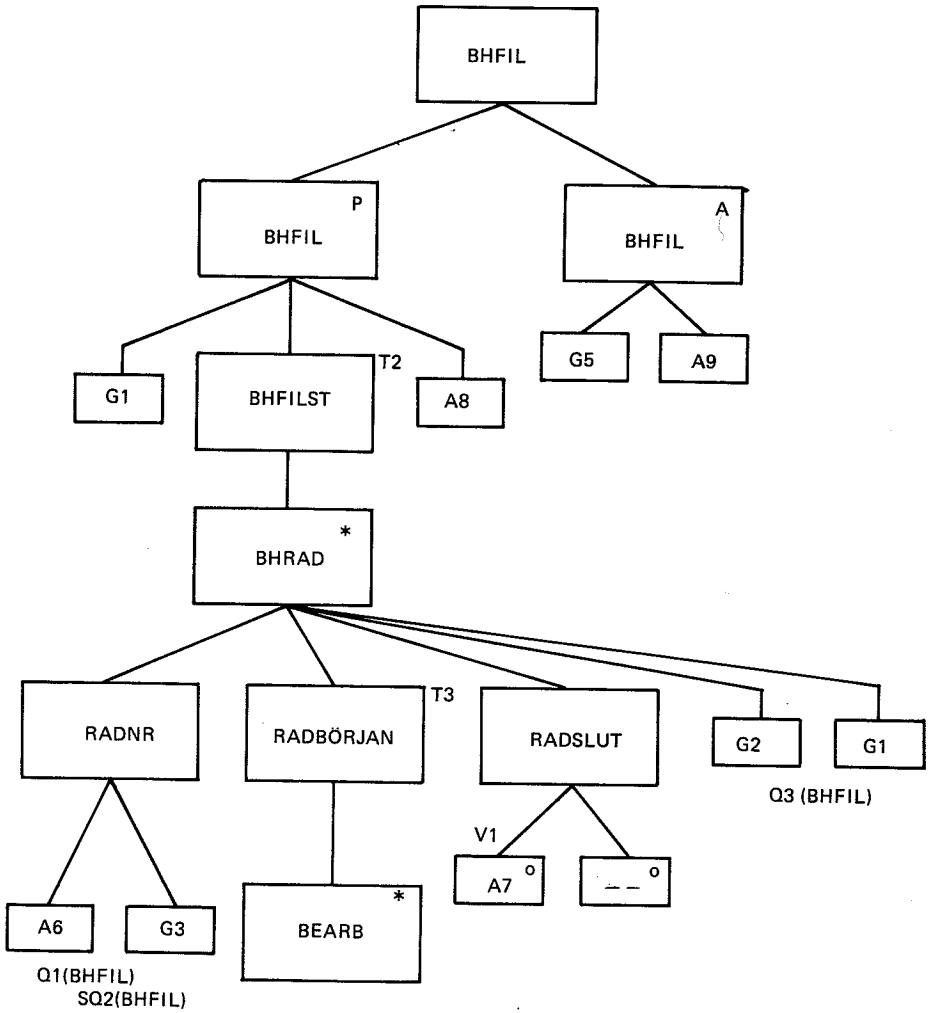
A1 Visa "Program som lägger %-tecken på
flyttalsvariabler och konstanter.
Fungerar på textfiler (.BAS)."

A2 Fråga efter infil. Öppna denna som fil 1.
Vid fel: Skriv felmeddelande, gör om.

A3 Fråga efter utfil. Öppna denna som fil 2.
Vid fel: Skriv felmeddelande, gör om.

A4 Stäng fil 1 och 2.

A5 Fråga "Vill du fortsätta". Ta svar.



Termineringsvillkor:

T2 Filslut, E9=34.
T3 B=13 eller B$="REM" eller B$="DATA".

Selektionsvillkor:

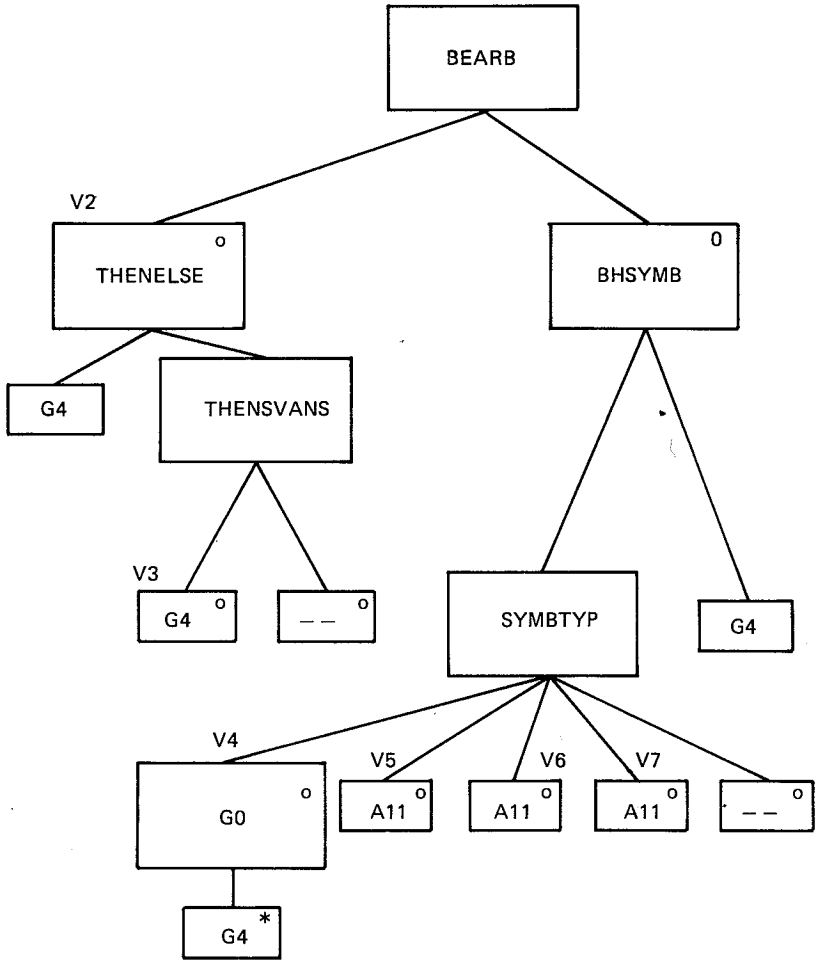
V1 B<>13 (Dvs "REM" eller "DATA").

Quits:

Q1 Om E9<>0 (Läsfel i G1).
SQ2 Om radnummer saknas. LET E9=16.
Q3 Om E9<>0 (Skrivfel i G2).

Operationer:

A6 Flytta radnummer från D1$ till D2$.
Visa D2$ på skärmen.
A7 Flytta resten av D1$ till D2$.
A8 Skriv "Jobbet klart" på skärmen.
A9 Skriv "Jobbet aborterat" på skärmen.



Selektionsvillkor:

V2 B$="THEN" eller B$="ELSE".
V3 Radnummer i B$.
V4 B$="GOTO", "GOSUB", "RESTORE" eller "ONERRORGOTO".
V5 En bokstav i B$.
V6 En bokstav + en siffra i B$.
V7 Konstant i B$ ej följd av %.

Operationer:

A11 D2$=D2$+"%".

4. GENERELLA SUBRUTINER

G1 Läsrad

Läser en post från fil nr 1.
Utvariabler:
D1$ Läst post.
E9 Eventuell felkod.

G2 Skrivrad

Skriver en post på fil nr 2.
Invariabel:
D2$ Post som ska skrivas.
Utvariabel:
E9 Eventuell felkod.

G3 Lässymb

Läser första symbol från inrad.
Invariabel:
D1$ Inrad.
Utvariabler:
Se G4.
Anrop:
G4 Nysymb.

G4 Nysymb

Läser nästa symbol från inrad. Lästa tecken överflyttas till utrad. Tecken som inte är Cr, ":", bokstav eller siffra samt stränguttryck läses först förbi.
Därefter tas alla tecken som är bokstav, siffra, "$" eller "%" som symbol.
Invariabel:
D1$ Inrad.
Utvariabler:
D1$ Resterande del av inrad.
D Ascii-kod för senast läst tecken.
D2$ Färdig del av utrad.
B$ Symbol.
B ASC(B$) eller 13 (Cr).
Anrop:
Cita.

G5 Fel

Skriver felmeddelande i form av felnummer.
Invariabel:
E9 Felnummer.

5. KORSREFERENSTABELL

Subrutiner / Variabler

	Läsrad	Skrivrad	Lässymb	Nysymb	Fel	Cita
B				U		
D			X	IU		IU
D1$	U		IU	IU		IU
D2$		I	IU	IU		IU
E9	U	U			I	
K						X
Anrop				Cita		

I=Invariabel U=Utvariabel X=Temporär variabel

PROCENT.BAS

```
100 ; CHR$(12)´¦ PROCENT ¦´
101 DIM B$=120,D1$=120,D2$=120
102 REM ----------------------------
103 REM - D1$  rad från infil
104 REM - D2$  rad till utfil
105 REM - B$   aktuell symbol
106 REM - B    asc(B$) eller cr
107 REM - D    senast lästa tkn.
108 REM - E9   felkod
109 REM ----------------------------
110 REM ¦ Huvudprogram
111 REM ----------------------------
112 REM -
113 REM * procent-seq *
114 ; : ; ´Program som lägger på %-tecken på´
115 ; ´flyttalsvariabler och konstanter.´
116 ; ´Fungerar på textfiler (.BAS).´
117 REM * prog-ite * (T1)
118 REM * job-seq *
119 REM * infil-seq *
120 ONERRORGOTO 124
121 ; : ; ´Infil ´; : INPUT B$
122 OPEN B$ ASFILE 1
123 GOTO 127
124 E9=ERRCODE
125 GOSUB 320 : REM * fel *
126 GOTO 120
127 REM * infil-end *
128 REM * utfil-seq *
129 ONERRORGOTO 133
130 ; : ;´Utfil ´; : INPUT B$
131 PREPARE B$ ASFILE 2
132 GOTO 136
133 E9=ERRCODE
134 GOSUB 320 : REM * fel *
135 GOTO 129
136 REM * utfil-end *
137 E9=0 : ONERRORGOTO 0
138 ;
139 GOSUB 153 : REM * bhfil *
140 CLOSE 1 : CLOSE 2
141 ; : ; ´Vill du fortsätta? (J/N)´; : GET B$
142 IF (ASC(B$) AND 223)<>ASC(´J´) THEN 145
143 REM * job-end *
```

```
144 GOTO 117
145 REM * prog-end *
146 REM * procent-end *
147 END
148 ;
149 REM ----------------------------
150 REM ¦ Behandla fil
151 REM ----------------------------
152 REM -
153 REM * bhfil-pos * (OK)
154 GOSUB 237 : REM * läsrad *
155 REM * bhfilst-ite * (filslut E9=34)
156 IF E9=34 THEN 223
157 REM * bhrad-seq *
158 IF E9<>0 THEN 226 : REM Q1(bhfil)
159 REM * radnr-seq *
160 K=INSTR(1,D1$,´ ´)
161 IF K=0 THEN E9=16 : GOTO 226 : REM * SQ2(bhfil)
162 D2$=LEFT$(D1$,K-1)
163 ; ´Rad nr:´D2$+CHR$(13);
164 D1$=RIGHT$(D1$,K)
165 GOSUB 266 : REM * lässymb *
166 REM * radnr-end *
167 REM * radbörjan-ite * (Cr,REM,DATA)
168 IF B=13 OR B$=´REM´ OR B$=´DATA´ THEN 212
169 REM * bearb-sel * (THEN,ELSE)
170 IF B$<>´THEN´ AND B$<>´ELSE´ THEN 180
171 REM * thenelse-seq *
172 GOSUB 276 : REM * nysymb *
173 REM * thensvans-sel * (radnr)
174 IF B<48 OR B>57 THEN 176
175 GOSUB 276 : REM * nysymb *
176 REM * thensvans-or *
177 REM * thensvans-end *
178 REM * thenelse-end *
179 GOTO 210
180 REM * bearb-or *
181 REM * bhsymb-seq *
182 REM * symbtyp-sel *
183 REM (GOTO,GOSUB,RESTORE,ONERRORGOTO)
184 IF LEN(B$)<4 THEN 192
185 IF INSTR(1,´GOTGOSRESONE´,LEFT$(B$,3))=0 THEN 192
186 REM * go-ite * (kol,Cr)
187 IF B=58 OR B=13 THEN 190
```



```
188 GOSUB 276 : REM * nysymb *
189 GOTO 186
190 REM * go-end *
191 GOTO 207
192 REM * symbtyp-or1 * (bokstav)
193 IF B<65 OR LEN(B$)<>1 THEN 196
194 D2$=D2$+´%´
195 GOTO 207
196 REM * symbtyp-or2 * (bokstav siffra)
197 IF LEN(B$)<>2 THEN 202
198 K=ASC(RIGHT$(B$,2))
199 IF B<65 OR K<48 OR K>57 THEN 202
200 D2$=D2$+´%´
201 GOTO 207
202 REM * symbtyp-or3 * (konstant)
203 IF B<48 OR B>57 THEN 206
204 IF RIGHT$(B$,LEN(B$))=´%´ THEN 206
205 D2$=D2$+´%´
206 REM * symbtyp-or4 *
207 REM * symbtyp-end *
208 GOSUB 276 : REM * nysymb *
209 REM * bhsymb-end *
210 REM * bearb-end *
211 GOTO 167
212 REM * radbörjan-end *
213 REM * radslut-sel * (not Cr)
214 IF B=13 THEN 216
215 D2$=D2$+CHR$(D)+LEFT$(D1$,LEN(D1$)-2)
216 REM * radslut-or *
217 REM * radslut-end *
218 GOSUB 252 : REM * skrivrad *
219 IF E9<>0 THEN 226 : REM Q3(bhfil)
220 GOSUB 237 : REM * läsrad *
221 REM * bhrad-end *
222 GOTO 155
223 REM * bhfilst-end *
224 ; : ; ´Jobbet klart.´
225 GOTO 229
226 REM * bhfil-adm * (fel)
227 GOSUB 320 : REM * fel *
228 ; : ; ´Jobbet aborterat´
229 REM * bhfil-end *
230 RETURN
231 ;
```

```
232 REM ----------------------------
233 REM ¦ G1 Läsrad
234 REM ----------------------------
235 REM - läs en rad från infilen.
236 REM -
237 REM * läsrad *
238 E9=0
239 ONERRORGOTO 243
240 INPUTLINE #1,D1$
241 ONERRORGOTO 0
242 GOTO 245
243 E9=ERRCODE
244 REM * läsrad-end *
245 RETURN
246 ;
247 REM ----------------------------
248 REM ¦ G2 Skrivrad
249 REM ----------------------------
250 REM - Skriv en rad på utfilen.
251 REM -
252 REM * skrivrad *
253 E9=0
254 ONERRORGOTO 258
255 PRINT #2,D2$
256 ONERRORGOTO 0
257 GOTO 260
258 E9=ERRCODE
259 REM * skrivrad-end *
260 RETURN
261 ;
262 REM ----------------------------
263 REM ¦ G3 Läs första symbol
264 REM ----------------------------
265 REM -
266 REM * lässymb *
267 D=ASC(D1$) : D1$=RIGHT$(D1$,2)
268 GOSUB 276 : REM * nysymb *
269 REM * lässymb-end *
270 RETURN
271 ;
272 REM ----------------------------
273 REM ¦ G4 Läs symbol
274 REM ----------------------------
275 REM -
```



```
276 REM * nysymb-seq *
277 REM * separator-ite *
278 REM (cr,kol,bokstav,siffra)
279 IF D=13 THEN 286
280 IF D>64 OR (D>47 AND D<59) THEN 286
281 IF D<>34 AND D<>39 THEN 283
282 GOSUB 307 : REM * cita *
283 D2$=D2$+CHR$(D)
284 D=ASC(D1$) : D1$=RIGHT$(D1$,2)
285 GOTO 277
286 REM * separator-end *
287 B$=""
288 IF D=13 THEN B$=CHR$(13)
289 REM * symbol-ite *
290 REM (ej bokstav,siffra,$,%, :    )
291 IF D<48 AND (D<36 OR D>37) THEN 297
292 IF D>58 AND D<65 THEN 297
293 D2$=D2$+CHR$(D)
294 B$=B$+CHR$(D)
295 D=ASC(D1$) : D1$=RIGHT$(D1$,2)
296 GOTO 289
297 REM * symbol-end *
298 B=ASC(B$)
299 REM * nysymb-end *
300 RETURN
301 ;
302 REM ----------------------------
303 REM ¦ Citation
304 REM ----------------------------
305 REM - Passera stränguttryck
306 REM -
307 REM * cita *
308 K=INSTR(1,D1$,CHR$(D))
309 D2$=D2$+CHR$(D)+LEFT$(D1$,K-1)
310 D1$=RIGHT$(D1$,K)
311 D=ASC(D1$) : D1$=RIGHT$(D1$,2)
312 REM * cita-end *
313 RETURN
314 ;
315 REM ----------------------------
316 REM ¦ G5 Felmeddelande
317 REM ----------------------------
318 REM - Skriv felmeddelande
319 REM -
```

PROCENT.BAS

```
320 REM * fel *
321 ; : ; ´Fel nr´E9´ (Se fellista)´
322 REM * fel-end *
323 RETURN
```


Appendix G
Rutiner för datalagring på kassettband

KASSETT2.BAS

```
1000 REM ------------------------------
1010 REM -
1020 REM ¦ Subrutin som lagrar D2$ i
1030 REM - D1$. Varje gång D1$ blir full
1040 REM - spelas den ut på kassetten.
1050 REM -
1060 REM -In    D2$ = ´data som ska skrivas´
1070 REM -
1080 REM -Glob  D1$
1090 REM -
1100 REM ------------------------------
1110 REM * PRINT *
1120 IF LEN(D1$)+LEN(D2$)>253% THEN 1150
1130 D1$=D1$+D2$+CHR$(13%)
1140 GOTO 1250
1150 REM
1160 REM Spela ut på band
1170 REM
1180 GOSUB 1940 : REM Starta motor
1190 GOSUB 1450 : REM skriv D1$
1200 REM
1210 REM Stoppa motorn
1220 REM
1230 OUT 58%,INP(58%) AND 223%
1240 D1$=D2$+CHR$(13%)
1250 RETURN
1260 REM
1270 REM ------------------------------
1280 REM -
1290 REM ¦ Skapa kassettfil
1300 REM -
1310 REM -In    DO$ = filnamn
1320 REM -      DO% = filnummer
1330 REM -
1340 REM ------------------------------
1350 REM * PREPARE *
1360 PREPARE DO$ ASFILE DO%
1370 REM
```

```
1380 REM Stoppa motorn
1390 REM
1400 OUT 58%,INP(58%) AND 223%
1410 REM
1420 D1%=PEEK(65021) : REM blocknummer
1430 RETURN
1440 REM
1450 REM ------------------------------
1460 REM -
1470 REM ¦ Subrutin för att skriva på
1480 REM - kassett med blockmellanrum
1490 REM -
1500 REM -In     DO% = filnummer
1510 REM -       D1$ = data att skriva
1520 REM -
1530 REM ------------------------------
1540 REM * SKRIV *
1550 PRINT #DO%,D1$
1560 REM
1570 REM Testa om dags för fördröjning
1580 REM
1590 IF PEEK(65021)<>D1% THEN GOSUB 1620
1600 REM
1610 RETURN
1620 REM ------------------------------
1630 REM - Subrutin för fördröjning
1640 REM ------------------------------
1650 FOR Z=0 TO 1500 : NEXT Z
1660 D1%=PEEK(65021) : REM nytt block
1670 RETURN
1680 REM
1690 REM ------------------------------
1700 REM -
1710 REM ¦ Stäng kassettfilen
1720 REM -
1730 REM -In     DO% = filnummer
1740 REM -
1750 REM ------------------------------
1760 REM * CLOSE *
1770 REM
1780 REM Se först till att hela D1$ är
1790 REM utskriven på kassetten.
1800 REM
1810 GOSUB 1940 : REM starta motorn
```



```
1820 GOSUB 1450 : REM skriv D1$
1830 REM Fyll ut med NULLtecken tills
1840 REM ett block har spelats ut.
1850 REM
1860 IF D1%<>PEEK(65021) THEN 1910
1870 PRINT #DO%,CHR$(0%); : GOTO 1860
1880 REM
1890 REM Stäng sedan filen
1900 REM
1910 CLOSE DO%
1920 RETURN
1930 REM
1940 REM ------------------------------
1950 REM -
1960 REM ¦ Starta motor och vänta lite
1970 REM -
1980 REM ------------------------------
1990 REM * STARTA *
2000 OUT 58%,INP(58%) OR 32%
2010 FOR Z=0 TO 500 : NEXT Z
2020 RETURN
2030 REM
2040 REM ------------------------------
2050 REM - Subrutiner för läsning
2060 REM ------------------------------
2070 REM -
2080 REM ¦ Öppna kassettfil
2090 REM -
2100 REM -In     DO$ = filnamn
2110 REM -       DO% = filnummer
2120 REM -
2130 REM -Ut     E9  = ev. felnummer
2140 REM -
2150 REM ------------------------------
2160 REM * OPEN *
2170 E9=0 : ONERRORGOTO 2200
2180 OPEN DO$ ASFILE DO%
2190 GOTO 2210
2200 E9=ERRCODE
2210 REM stoppa motorn
2220 OUT 58%,INP(58%) AND 223%
2230 RETURN
2240 REM
2250 REM ------------------------------
```

KASSETT2.BAS

```
2260 REM -
2270 REM ¦ Läs in en sträng från kass.
2280 REM -
2290 REM -In     DO% = filnummer
2300 REM -
2310 REM -Ut     D1$ = den lästa strängen
2320 REM -       E9  = ev. felnummer
2330 REM -
2340 REM ------------------------------
2350 REM * INPUTL *
2360 E9=0 : ONERRORGOTO 2410
2370 REM starta motorn
2380 OUT 58%,INP(58%) OR 32%
2390 INPUTLINE #DO%,D1$
2400 GOTO 2420
2410 E9=ERRCODE
2420 REM stoppa motorn
2430 OUT 58%,INP(58%) AND 223%
2440 RETURN
2450 REM
2460 REM ------------------------------
```


Appendix H
Rutiner för datalagring på diskett

DIREKT2.BAS

```
20000 REM ------------------------------
20010 REM ¦ DIREKT
20020 REM ------------------------------
20030 REM - Rutiner för läsning och
20040 REM - skrivning på direktfiler.
20050 REM - Varje block förutsätts inne-
20060 REM - hålla ett helt antal poster.
20070 REM -
20080 REM Dedicerade variabler.
20090 REM - DO%      Aktuellt filnummer (i).
20100 REM - D1%(i)   Akt. blocknr i fil i
20110 REM -          (Får ej ändras).
20120 REM - D2%(i)   Akt. postnr i fil i.
20130 REM - D3%(i)   Postlängd för fil i.
20140 REM - D4%(i)   Blockfaktor för fil i.
20150 REM -          (Ant. poster/block)
20160 REM - D5%(i)   Filstorlek (poster) för fil i.
20170 REM - D9%(i)   Skrivflagga
20180 REM -          (Får ej ändras).
20190 REM - DO$(i)   Filnamn för fil i+1.
20200 REM - D1$      Akt. post.
20210 REM - D1$(i)   Blockbuffer för fil i+1
20220 REM -          (Får ej ändras).
20230 REM -
20240 REM Temporära variabler.
20250 REM - P%
20260 REM -
20270 REM ------------------------------
20280 REM ¦ Filinit
20290 REM ------------------------------
20300 REM - Denna rutin måste först
20310 REM - anropas.
20320 REM - Varje fil definieras med en
20330 REM - DATAsats enl. exempel nedan.
20340 REM -
20350 REM -DATA <ant. filer>
20360 REM -DATA <filnamn-1>,<postlängd>,<fillängd>
20370 REM -DATA <filnamn-2>,<postlängd>,<fillängd>
```

```
20380 REM -...
20390 REM -
20400 REM *Filinit*
20410 READ P%
20420 DIM DO$(P%-1%)=16%,QO$=253%,D1$(P%-1%)=253%
20430 DIM D1%(P%),D2%(P%),D3%(P%),D4%(P%),D5%(P%)
20440 D9%(0%)=0%
20450 FOR DO%=1% TO P%
20460 D1%(DO%)=9999%
20470 READ DO$(DO%-1%) : READ D3%(DO%)
20480 D4%(DO%)=253%/D3%(DO%) : READ D5%(DO%)
20490 IF D3%(DO%)>D9%(0%) THEN D9%(0%)=D3%(DO%)
20500 NEXT DO%
20510 DIM D1$=D9%(0%) : REM Största postlängd
20520 REM *Filinit-end*
20530 RETURN
20540 ;
20550 REM ------------------------------
20560 REM ¦ Preparera
20570 REM ------------------------------
20580 REM - Fil prepareras till aktuell
20590 REM - storlek, fylls med nollor.
20600 REM Invariabler.
20610 REM - DO%  Akt. filnummer.
20620 REM - D5%  Filstorlek (poster).
20630 REM Utvariabler.
20640 REM - E9%  Ev. felkod.
20650 REM ------------------------------
20660 REM *Prep*
20670 E9%=0% : ONERRORGOTO 20750
20680 PREPARE DO$(DO%-1%) ASFILE DO%
20690 D1%(DO%)=0% : D1$(DO%-1%)=STRING$(253%,0%)
20700 GOSUB 21730 : REM *Skrivblock*
20710 IF E9% THEN 20740
20720 D1%(DO%)=D1%(DO%)+1%
20730 IF D1%(DO%)<=D5%(DO%)/D4%(DO%) THEN 20700
20740 CLOSE DO% : ONERRORGOTO 0 : GOTO 20760
20750 E9%=ERRCODE
20760 REM *Prep-end*
20770 RETURN
20780 ;
20790 REM ------------------------------
20800 REM ¦ Öppna
20810 REM ------------------------------
```



```
20820 REM - Fil öppnas för skrivning
20830 REM - och/eller läsning.
20840 REM Invariabler.
20850 REM - DO%  Akt. filnummer.
20860 REM Utvariabler.
20870 REM - E9%  Ev. felkod.
20880 REM ------------------------------
20890 REM *Öppna*
20900 E9%=0% : ONERRORGOTO 20950
20910 OPEN DO$(DO%-1%) ASFILE DO%
20920 ONERRORGOTO 0
20930 D9%(DO%)=0% : D1%(DO%)=9999%
20940 GOTO 20960
20950 E9%=ERRCODE
20960 REM *Öppna-end*
20970 RETURN
20980 ;
20990 REM ------------------------------
21000 REM ¦ Stäng
21010 REM ------------------------------
21020 REM - Aktuell fil stängs.
21030 REM Invariabler.
21040 REM - DO%  Akt. filnummer.
21050 REM Utvariabler.
21060 REM - E9%  Ev. felkod.
21070 REM Anrop.
21080 REM -Skrivblock
21090 REM ------------------------------
21100 REM *Stäng*
21110 IF D9%(DO%) THEN GOSUB 21730 : REM *Skrivblock*
21120 CLOSE DO%
21130 REM *Stäng-end*
21140 RETURN
21150 ;
21160 REM ------------------------------
21170 REM ¦ Skrivpost
21180 REM ------------------------------
21190 REM - En post skrivs i buffert.
21200 REM Invariabler.
21210 REM - DO%       Filnummer
21220 REM - D2%(DO%)  Postnummer.
21230 REM - D1$  Post som ska skrivas.
21240 REM Utvariabler.
21250 REM - E9%  Ev. felkod.
```

```
21260 REM Anrop.
21270 REM -Skrivblock
21280 REM -Läsblock
21290 REM ----------------------------
21300 REM *Skrivpost*
21310 E9%=0%
21320 IF LEN(D1$)=D3%(D0%) THEN 21340
21330 E9%=100% : GOTO 21450
21340 IF D2%(D0%)/D4%(D0%)=D1%(D0%) THEN 21400
21350 IF D9%(D0%) THEN GOSUB 21730 : REM *Skrivblock*
21360 IF E9% THEN 21450
21370 D1%(D0%)=D2%(D0%)/D4%(D0%)
21380 GOSUB 21830 : REM *Läsblock*
21390 IF E9% THEN 21450
21400 P%=(D2%(D0%)-D1%(D0%)*D4%(D0%))*D3%(D0%)
21430 D1$(D0%-1%)=LEFT$(D1$(D0%-1%),
      P%)+D1$+RIGHT$(D1$(D0%-1%),P%+D3%(D0%)+1%)
21440 D9%(D0%)=-1%
21450 REM *Skrivpost-end*
21460 RETURN
21470 ;
21480 REM ----------------------------
21490 REM ¦ Läspost
21500 REM ----------------------------
21510 REM - En post läses från buffert.
21520 REM - D0%       Filnummer
21530 REM - D2%(D0%)  Postnummer.
21540 REM Utvariabler.
21550 REM - D1$  Läst post.
21560 REM - E9%  Ev. felkod.
21570 REM Anrop.
21580 REM -Skrivblock
21590 REM -Läsblock
21600 REM ----------------------------
21610 REM *Läspost*
21620 IF D2%(D0%)/D4%(D0%)=D1%(D0%) THEN 21680
21630 IF D9%(D0%) THEN GOSUB 21730 : REM *Skrivblock*
21640 IF E9% THEN 21700
21650 D1%(D0%)=D2%(D0%)/D4%(D0%)
21660 GOSUB 21830 : REM *Läsblock*
21670 D9%(D0%)=0%
21680 P%=(D2%(D0%)-D1%(D0%)*D4%(D0%))*D3%(D0%)
21690 D1$=MID$(D1$(D0%-1%),P%+1%,D3%(D0%))
```



```
21700 REM *Läspost-end*
21710 RETURN
21720 ;
21730 REM *Skrivblock*
21740 E9%=0% : ONERRORGOTO 21780
21750 P%=CALL(28666%,D0%) : Q0$=D1$(D0%-1%)
21760 P%=CALL(28670%,D1%(D0%))
21770 GOTO 21790
21780 E9%=ERRCODE
21790 ONERRORGOTO 0
21800 REM *Skrivblock-end*
21810 RETURN
21820 ;
21830 REM *Läsblock*
21840 E9%=0% : ONERRORGOTO 21880
21850 P%=CALL(28666%,D0%)+CALL(28668%,D1%(D0%))
21860 D1$(D0%-1%)=Q0$
21870 GOTO 21890
21880 E9%=ERRCODE : D1$(D0%-1%)=STRING$(253%,0%)
21890 ONERRORGOTO 0
21900 REM *Läsblock-end*
21910 RETURN
```

Appendix I
Rekommenderade kortadresser till I/O—kort

--ADRESS---				--ADRESS---			
Hex	Dec	ASC	Kort	Hex	Dec	ASC	Kort
00	0	0	-	20	32	P	-
01	1	1	Printer	21	33	Q	-
02	2	2	PIO-kort	22	34	R	-
03	3	3	PIO-kort	23	35	S	-
04	4	4	PIO-kort	24	36	T	-
05	5	5	PIO-kort	25	37	U	-
06	6	6	-	26	38	V	-
07	7	7	-	27	39	W	-
08	8	8	-	28	40	X	-
09	9	9	-	29	41	Y	-
0A	10		SIO-kort	2A	42	Z	-
0B	11		SIO-kort	2B	43	Ä	-
0C	12		SIO-kort	2C	44	Ö	-
0D	13			2D	45		RESERVERAD
0E	14			2E	46		FLEXSKIVEENHET
0F	15			2F	47		
10	16			30	48		
11	17	A	A-D-kort	31	49		IEC-buss (IEEE488)
12	18	B	A-D-kort	32	50		
13	19	C	D-A-kort	33	51		
14	20	D	D-A-kort	34	52		
15	21	E	D-A-kort	35	53		
16	22	F	D-A-kort	36	54		
17	23	G	-	37	55		
18	24	H	-	38	56		
19	25	I	-	39	57		
1A	26	J	-	3A	58		
1B	27	K	-	3B	59		
1C	28	L	-	3C	60		P-40-printer
1D	29	M	-	3D	61		
1E	30	N	-	3E	62		
1F	31	O	-	3F	63		

Hex är kortadressen i hexadecimal form.
Dec är kortadressen i decimal form.
ASC är ASCII-tecknet för kortval i en del färdiga drivrutiner till I/O-kort, såsom drivrutiner för parallell BCD till PIO-kortet och AD-rutiner för A-D-kortet.

De adresser som används i samband med färdiga drivrutiner, är markerade med ett streck (-) om inget kort är rekommenderat dit.

Vid val av kortadress väljs i första hand de som är angivna. I andra hand används adresser i den vänstra listan (0-31). Adresser efter 44 används i första hand till fasta utrustningar från tillverkaren.

Sakregister

Anders Isaksson – Örjan Kärrsgård Avancerad programmering på ABC80

Boken tar upp många av är nöd-
vändiga för ett effektivt ad pro-
grammering, dataelement ommu-
nikation. Många konkreta xempel
för anslutning av yttre dat
Förutom den löpande te... diserade
normer och i ABC80 använda kopplingar, karaktärer, koder m m. En hel del information som tidigare aldrig redovisats i samlad form m a o.

. . . Avancerad programmering på ABC80 kommer att fylla en viktig lucka i den tidigare utgivna informationen om ABC80. Boken är ett så pass värdefullt dokument, att envar med intresse för egen programutveckling bör skaffa sig ett exemplar. Jag är övertygad om att de flesta ABC80-användare har något att hämta ur den matnyttiga skriften.

Bengt Olwig, Datornytt

Dataserien – datalogi och programmering

J Aldenheim – C Anlér – U Warmlöv	Introduktion till minidatorn
G Birtwistle – O-J Dahl – B Myhrhaug – K Nygaard	SIMULA BEGIN
O Bjorner	Avancerad COBOL
O Bjorner – K Holm	Grunderna i COBOL
J Bohman – C-E Froberg	Programmera i APL
B Christensen	RUN COMAL 1 – strukturerad BASIC
T Ekman	Programmering i ALGOL 60
T Ekman – G Eriksson	Programmering i Fortran 77
T Ekman – G Eriksson	Programmering i Standard FORTRAN
L Ewald – E Roupé – B Wahlqvist	Lärobok i BASIC
S Forster m fl	Assembler för UNIVAC 1100
B Frejhagen	Strukturerad programmering
S Frenkel – S Persson	Elementar FORTRAN
A Haraldsson	Programmering i PASCAL
M Hedenborg – G Sundberg	ADB – BASIC
L Ingevaldsson	JSP – en praktisk metod för programkonstruktion
A Isaksson – Ö Kärrsgård	Avancerad programmering på ABC80
S Kallin	Lärobok i FORTRAN
S Kallin – C Odén	Lärobok i PL/1
S-E Karlsson – B Upman	Lärobok i RPG II
H Lunell	Datalogi – en inledande översikt
A Lysegård	Assembler för UNIVAC 1108
A Lysegård	Lärobok i COBOL
Miniforsk	Uppsala BASIC för ALPHA LSI
S Nachmens	Datasystem och datorsystem
P Naur	Concise Survey of Computer Methods
J Palme	Programmeringsspråk
E Roupé	Programmeringsövningar i BASIC
U-G Arlehag	Lärobok i Assembler

Studentlitteratur – ett förlag
inom Utbildningshuset