LUXOR Datorer

BIT FÖR BIT MED ABC 800

ABC 800®

FÖRORD

Detta dokument riktar sig till dig som redan kan en hel del om BASIC- och ASSEMBLER-programmering och som vill veta mer om möjligheterna för avancerad programmering på ABC 800.

Du som vill lära dig grundläggande BASIC-programmering bör läsa "BASIC II Boken", utgiven av Liber.

Eftersom systemprogramvaran, liksom även maskinvaran, ständigt utvecklas, garanterar Luxor ej att innehållet i detta dokument till alla delar överensstämmer med den dator som programmen utvecklas på.

Materialet är framtaget av NANCO Elektronik, Henån.

INNEHÅLL

BILAGOR

1.INLEDNING

För att uppnå förståelse och enighet mellan användare, som refererar till denna skrifts programlistor och exempel, följer alla listningar och exempel en standard som redovisas nedan.

INTEGER-mode	Används konsekvent i alla exempel. Förekommer flyttalsvariabler noteras dessa med suffixet punkt (.). Ex: A.=Flyt.(8)+13.45
EXTEND-mode	Används konsekvent i alla exempel.
OPTION BASE 0	Används konsekvent i alla exempel.
SINGLE	Används konsekvent i alla exempel.
Ec	Används som global felvariabel och motsvarar efter funktionsanrop eventuell error-kod (ERRCODE) i anropad funktion.
Z	Används som "dummy"-variabel vid anrop av funktioner som returnerar integer-värde.

2. BASIC-PROGRAMMERING

2.1 BASIC II

2.1.1 Skillnader i BASIC mellan ABC 800 och ABC 80

BASIC II till ABC800 är en utökning av ABC80 BASIC. BASIC II är i de flesta fall kompatibel med ABC80 BASIC. I en del fall förekommer vissa skillnader som redovisas nedan. Se även BASIC II manualen.

De skillnader som förekommer kan indelas med avseende på orsak.

* Skillnader i avsikt att uppfylla ANSI-standard för BASIC.
  (Punkt 1 - 5)

* Rättningar av felaktigheter i ABC80 BASIC.
  (Punkt 6 - 7)

* Utökningar och förbättringar.
  (Punkt 8 - 24)

1. Vid tilldelning av flyttalsvärde till heltalsvariabel sker avrundning.

   Exempel: A%=3.567 Varibeln A% tilldelas värdet:
   ABC800: 4 ABC80: 3

2. Vid utskrift med hjälp av TAB anges utskriftsposition från TAB(1) på ABC800.

   Exempel: ;TAB(5)´B´ Utskrift sker i position:
   ABC800: 5 (dvs pos. 1-40/80) ABC80: 6 (dvs pos. 0-39)

3. Instruktionen INPUT får innehålla en ledtext som skrivs ut. Skriver man INPUT A,B skall värden matas in åtskiljda av komma, i annat fall ges felmeddelande.
   Om man skriver INPUT A, kommer cursorn att stå i nästa kolumn efter inmatningen.
   OBS!
   Använd endast denna funktion då inmatning sker från tangentbordet.

4. Blanktecken i DATA-satser behöver inte stå inom delimeters. Undantag från denna regel är inledande och/eller avslutande blanktecken.

5. Det är tillåtet att ge blanktecken i INPUT-satsen. Observera dock att INPUT eliminerar inledande och/eller avslutande blanktecken.

6. INSTR ger ej fel vid den tomma strängen. ABC80 kunde ge negativa värden.

7. Kolon (:) i DATA och REM ger ej de extra blanktecken vid LIST som felaktigt skapades i ABC80.

8. Vid utskrift av numeriska variabler åtskiljda av semikolon (;) läggs ett extra blanktecken in mellan talen.

   Exempel: ;X;Y Detta ger följande utskrift:
   ABC800: 0 0   ABC80: 0 0

9. Vid konvertering från numeriskt värde med hjälp av NUM¤ ges inget inledande blanktecken vid positivt värde.

   Exempel: 20 I=1234
            30 ;NUM¤(I) Ger följande utskrift:
   ABC800:1234   ABC80: 1234

10. Vid beräkning med ASCII-aritmetik är antalet siffror utökade till 125 och exponent tillåts som invärde.

11. Vid dimensionering av variabler med DIM är det tillåtet att ge både undre och övre indexgräns, dvs DIM A(2:16) är tillåtet. Dessutom tillåts valfritt antal dimensioner.

12. Ett alternativ till REM finns, nämligen utropstecken (!). Skrivs den på samma rad som en annan instruktion behöver den ej föregås av kolon (:).

13. Instruktionen ON ERROR GOTO 0 ersätts med ON ERROR GOTO.

14. Instruktionen END skall stå ensam på raden. END stänger alla filer men nollställer ej variabler.

15. Läsning (pollning) av tangentbordet med INP(56) finns ej. Liknande funktion kan erhållas med SYS(5) funktionen.

16. CALL-funktionerna vid direkt access på ABC80 ersätts med POSIT,PUT,GET...COUNT.

    Exempel:
    Läsning:
    ABC800: POSIT#filnr,sektornr*253:GET#filnr,Q0¤ COUNT 253
    ABC80: Z=CALL(28666,filnr)+CALL(28668,sektornr)
    Skrivning:
    ABC800: POSIT#filnr,sektornr*253:PUT#filnr,A¤
    ABC80: Z=CALL(28666,filnr):Q0¤=A¤:Z=CALL(28670,sektornr)

17. Instruktionen CLOSE utan något filnr stänger alla filer. Vill man ge mer än ett filnr är detta också tillåtet med satsen CLOSE fil1,fil2,fil3 ....

18. Instruktionen CHAIN ´´ ger ERROR 136. Den kan ersättas av antingen CHAIN ´NUL:´ eller END. CHAIN är även behandlingsbart med ON ERROR GOTO.

19. Vid kommandot LIST på fil är det tillåtet att ange radgränser.
    Exempel: LIST PR:,10000-20000 Listar raderna 10000-20000 på printern.

20. Vid kommandot REN är det tillåtet att ange ett intervall.
    Exempel: REN 10000,10,10- Omnumrerar raderna 10 - med startvärde 10000 och intervall 10.

21. Vid kommandot MERGE skall filen vara ett listat program (.BAS).

22. Utskrifter från TRACE kan ske på fil och ej bara på bildskärmen.

23. DEF FN fungerar på flyttal, heltal och strängar. Den kan även vara flerradig.

24. CHR¤ har ej enbart fyra argument längre utan valfritt antal. Man kan alltså skriva CHR¤(A,B,C,D,E,F,G).

2.1.2 Nyheter i BASIC II

Förutom de skillnader mot ABC80 BASIC som redovisades i förra kapitlet, har det naturligtvis tillkommit en hel del nyheter i BASIC II. En kortfattad beskrivning över dessa nyheter ges nedan. I ABC80 kolumnen finns en beskrivning (vid de tillfällen en motsvarighet existerar) av ersättningsförfarande. För ytterligare information av nyheter hänvisas till BASIC II manualen.

KOMMANDO	ABC800	ABC80
AUTO	Ger automatisk radnumrering.	---
CONTINUE CON	Startar exekvering av ett stoppat program vid den rad där programmet stoppades.	---
ERASE	Tar bort rader ur programmet.	
GOTO	Startar exekvering på en viss rad.	POKE 65060,0:GOTO rl

INSTRUKTION	ABC800	ABC80
COMMON	Deklaration av de variabler, vars värden skall överföras till ett annat program.	Använd POKE-arean eller mellanlagring på fil.
DEF FN ... RETURN uttryck FNEND	Flerradiga funktioner tillåts. Avgränsas med DEF FN och FNEND. Värdet returneras med instruktionen RETURN uttryck.	GOSUB /RETURN
DIGITS	Anger antalet siffror som skrivs ut med PRINT. Påverkar även kolumnbredden. Används ej DIGITS i programmet är kolumnbredden vid SINGLE = 6 och vid DOUBLE = 16.	---
DOUBLE	Ändrar alla variabler och uttryck, som är flyttal,till dubbel precision (16 siffror).	---
EXTEND	Anger att det är tillåtet att använda långa variabelnamn.	---
FLOAT	Anger att alla variabler, konstanter och funktioner antas vara flyttalsvariabler om ej suffix anges.	---
INTEGER	Anger att alla variabler, konstanter och funktioner antas vara heltalsvariabler om ej suffix anges.	---
NO EXTEND	Anger att det är otillåtet att använda långa variabelnamn.	---
OPTION BASE	Anger undre gräns för vektorindex. Tillåtna värden är 0 och 1, med defaultvärde 0.	---

INSTRUKTION	ABC800	ABC80
PRINT USING	Skriver ut tal och strängar enligt ett format som bestäms av användaren. Ex. PRINT USING "###.##" A.	Formatera utskriften med en subrutin.
RESUME	Anger vart återhopp skall ske då ett fel, omhändertaget av ON ERROR GOTO, sker. Utelämnas radnummer vid RESUME sker återhoppet till den rad där felet uppstod.	---
SINGLE	Ändrar alla variabler och uttryck,som är flyttal,till enkel precision (7 siffror).	---
WHILE uttryck ... WEND	Satserna mellan WHILE och WEND utförs så länge uttrycket efter WHILE är sant.	r1 IF NOT uttryck THEN GOTO r2 ..... GOTO r1 r2 ....

FUNKTION	ABC800	ABC80
CVT%¤	Lagrar ett heltal i en sträng.	A¤=CHR¤(A% AND 255%,A%/256%)
CVT¤%	Återskapar ett heltal ur en sträng.	A%=ASC(LEFT¤(A¤,1)) +ASC(RIGHT¤(A¤,1))* 256%
CVTF¤	Lagrar ett flyttal i en sträng.	---
CVT¤F	Återskapar ett flyttal ur en sträng.	---
HEX¤	Omvandlar ett heltal till en hexadecimal sträng.	Konstruera en subrutin som utför omvandlingen.
MOD	Ger resten vid heltalsdivision.	T%-INT(T%/N%)*N%
OCT¤	Omvandlar ett heltal till en oktal sträng.	Konstruera en subrutin som utför omvandlingen.
PEEK2	Läser innehållet i två (2) bytes.	PEEK(A%)+ PEEK(A%+1%)*256%
SYS	Ger systemstatus.	Läs av resp internvariabel med PEEK. Se ABC80 manualen.
TIME¤	Ger tidsangivelse på formen år-mån-dag tim.min.sek Ex. 1982-09-13 10:30:15	Läs av klockan med en programrutin. Se ABC80 manualen.
VAROOT	Ger adressen till en variabels plats i variabellistan.	Sök igenom variabellistan. Se AVANCERAD PROGRAMMERING PÅ ABC80.
VARPTR	Ger adressen till en variabels värde.	Sök igenom variabellistan. Se AVANCERAD PROGRAMMERING PÅ ABC80.

2.2 Programmera ABC 800

2.2.1 Generell programmeringsteknik

Detta avsnitt beskriver hur man bör gå till väga vid konstruktion av större programsystem.

ARBETSGÅNG

Vi skall här översiktligt gå igenom lämplig arbetsgång, för att senare steg för steg gå igenom de olika delarna mer i detalj.

1 DEFINIERA DATASTRUKTUR (diskett)

2 DEFINIERA DATASTRUKTUR (listor)

3 DEFINIERA DIALOG (skärmar/menyer..)

4 DEFINIERA PROGRAMMETS GROVSTRUKTUR

5 KODNING, PROGRAMGENERERING

DEFINIERA DATASTRUKTUR

Det första man skall göra är att bestämma vad programmet skall ha för in- och utdata. En lämplig konstruktionsmetod är därför JSP, en generell programkonstruktionsmetod som arbetar utifrån dessa datastrukturer. (För beskrivning av JSP - se kap 2.2.2.)

DEFINIERA LISTOR

Innebär att vi bestämmer alla utskrifter och listor som vi önskar från vårt program.

Exempel på listor är

LAGER-LISTA i ett lagerprogram
KONTO-LISTA i ett bokföringsprogram

Till vår hjälp har vi den tidigare definierade datastrukturen varifrån vi kan se vilka uppgifter vi har tillgängliga i systemet.

DEFINIERA DIALOG

Dialogbeskrivningen, som liksom listdefinitionen egentligen är en del av datastrukturdefinitionen, består av en förteckning av de skärmar (layouter på bildskärmen), menyer och formulär (inmatningsskärmar) som skall ingå i systemet.

Exempel på formulär är

REGISTRERA NY KUND
KONTERINGSRUTIN

PROGRAMMETS GROVSTRUKTUR

Man har samtidigt med definitionen av ovanstående delar fått en första grovstruktur på hur det färdiga programmet kommer att se ut.

Därefter fortsätter man att förfina programmet i steg, tills man har fått ett färdigt program som grundar sig på datastrukturen. Tekniken med stegvis förfining av programmet kallas "Top-down"-kodning.

KODNING, PROGRAMGENERERING

* Använd långa variabelnamn.

* Använd INTEGER - mode.

* Använd flerradiga funktioner istället för GOSUB (simulerade procedurer).

* Utnyttja TRUE/FALSE - funktioner.

* Följ "Lök-principen" (Kommunikation mellan nivåer).

* Utnyttja "Read Ahead".

* Följ ABC800 Metodhandbok (ref. 3).

* Långa variabelnamn

Att använda långa variabelnamn innebär inte att man skall använda namn som tar upp en halv bildskärmsrad. Man skall inte använda namn som:

Biblioteksektor(Bibliotekssektor)
Prisinklusivemomspåvaran
Bildkorrektionsfaktor2

Dessa "jättenamn" leder förr eller senare till stavfel med påföljande felsökning. Ett bättre namn för exempelvis Bibliotekssektor kan vara Bibsekt. Detta är lika lätt att förstå men med mycket mindre risk för felstavning.

En allmän regel för namngivningen är att namnen skall vara relevanta och vad längden beträffar kan man oftast klara sig med upp till 10 tecken.

Ett undantag från regeln att alltid använda långa variabelnamn är loop- och slaskvariabler.

För loopvariblerna räcker det oftast gott med namn som I eller J. Lämpliga bokstäver är I - N.

OBS!
Det är ej tillåtet att använda lokala variabler i en FOR-loop. Här måste man använda en global variabel eller, som altenativ, byta till en WHILE-loop. COMMON-variabel kan ej användas som loopräknare i FOR-NEXT loop.

För att få ett mer lättläst program kan man tex placera alla texter på ett ställe och inte ha dem utspridda över hela programmet.

* INTEGER-mode

Att man skall använda INTEGER-mode beror på att större delen av ditt program antagligen inte använder flyttal. I de fall flyttal används registreras dessa med punkt (.).

Fördelen med INTEGER-mode är att läsbarheten ökar då alla %-tecken försvinner och man riskerar ej att missa att sätta ut något %-tecken.

* Flerradiga funktioner

Varför man skall använda funktioner i stället för GOSUB beror på att GOSUB försvårar själva läsningen av programmet, medan funktioner med relevanta namn förenklar läsningen.

Exempel:

```
100 A = 1
110 GOSUB 300
120 A = 2
130 GOSUB 300
140 END
300 Z = CALL(24678,A)
310 RETURN
```

Bättre variant:

```
100 DEF FNReadsekt(I) = CALL(24678,I)
110 Z=FNReadsekt(1)
120 Z=FNReadsekt(2)
130 END
```

En god regel är att skriva så mycket som möjligt i funktionerna och att organisera dessa så att de ej ligger utspridda över hela programmet utan någon som helst ordning (struktur).

För att strukturera uppläggningen av funktionerna kan två olika metoder användas.

Antingen placeras alla funktioner, som har likartade funktioner, på ett ställe i programmet så att man får ett sk paket, tex filpaket,tangentbordspaket etc.
Eller så placeras de så att alla funktioner som finns på samma nivå i JSP-strukturen ligger på samma ställe.
Fördelen med att ha det organiserat på det senare sättet är att programstrukturen syns lättare. Fördelen med det förra sättet är då man bara är intresserad av delrutiner i programmet då samhörande funktioner finns på samma ställe. En annan fördel med detta placeringssätt är att man också får lättare att göra MERGE, om man till ett annat program bara är intresserad av ett speciellt paket.

* TRUE/FALSE funktioner

Dessa funktioner returnerar 0 eller -1 beroende på om funktionen utfördes korrekt eller om ett fel uppstod.

Exempel:

```
100 IF NOT FNFunktion THEN P=FNError("FEL")
```

I exemplet ovan skrivs felmeddelandet "FEL" ut om något fel uppstår under exekveringen av FNFunktion.

Allmänt kan man säga att funktionerna används när bara två tillstånd kan uppstå. Oftast rätt eller fel. Den anropande rutinen kan ta hand om de olika tillstånden och fortsätta bearbetningen beroende på returvärdet från funktionen.

* "Lök-principen"

"Lök-principen" går ut på att programmet (systemet) är uppbyggt i olika skal (nivåer) precis som en lök. Varje skal får bara kommunicera med skalet alldeles ovanför eller nedanför aktuellt skal, enligt något bestämt gränssnitt. Enligt "Lök-principen" ser ABC800-program ut så här:

Skal 1: Huvudprogram (huvudloop)

Skal 2: Funktioner

Skal 3: Assemblerrutiner

Skal 4: Hårdvara

"Lök-principen" kan tex användas vid konstruktion av assemblerrutiner, som BASIC skall kommunicera med. I detta fall bör anropet till assemblerrutinen ligga i en egen funktion.

Exempel:

```
10 DEF FNAsmcall(Adr,Arg) = CALL(Adr,Arg)
```

* "Read Ahead"

Denna teknik används vid olika slag av repetitioner (tex WHILE), men är mest använd i samband med filhantering.

"Read Ahead" går ut på att man placerar en läsoperation/tilldelning omedelbart efter öppnandet av filen, alternativt vid loopen alldeles före WHILE, och en läsoperation omedelbart efter det att man har använt posten, alternativt vid loopen alldeles före WEND.

Exempel: Loop. Jämförelse med FOR-NEXT.

```
10000 FOR I = 1 TO 10
10010   ! Bearbeta
10020 NEXT I
```

```
10000 I=1
10010 WHILE I<=10
10020   ! -- Bearbeta
10030   I=I+1
10040 WEND
```

Exempel: Enkelt filhanteringsexempel.

```
10000 INPUT #1,A
10010 WHILE A<>0
10020   ! Bearbeta A
10030   INPUT #1,A
10040 WEND
```

Exempel: Filhanteringsexempel med funktioner.

```
10000 DEF FNLinput¤(Fil) Local Text¤=160
10010   Ec=0 : ON ERROR GOTO 10040
10020   INPUT LINE #Fil,Text¤
10030   RETURN LEFT¤(Text¤,LEN(Text¤)-2)
10040   Ec=ERRCODE : RETURN ""
10050 FNEND
10060 !
10070 OPEN "LÄSFIL.TXT" AS FILE 1
10080 A¤=FNLinput¤(1)
10090 Eof=Ec<>0
10100 WHILE NOT Eof
10110   ! -- Bearbeta A¤
10120   A¤=FNLinput¤(1)
10130   Eof=Ec<>0
10140 WEND
.
.
```

Ett exempel på användning av "Read Ahead" är då en post skall bearbetas olika beroende på något villkor.

För att kunna testa villkoret måste läsning ske innan villkorssatsen utförs.

* ABC800 metodhandbok

Denna handbok är utgiven för att standardisera utformningen av dialogprogram etc för ABC800.

Att följa en standard är viktigt. Om alla följer standarden kommer alla program att se likadana ut för användaren, men det blir också lättare för programmerarna att följa varandras program.

Menyer - kommandostyrning

Dina program kan antingen vara kommando- eller frågeorienterade (menyer, löpande frågor osv). Ofta är det viktigt för användaren vilken uppläggning man väljer. Ett program får aldrig växelvis vara både kommando och frågeorienterat.

För ovana användare är menyuppläggningen bäst.

Viktigt är att menyerna är logiskt uppbyggda så att användaren inte skall behöva utföra onödigt sökande för att komma dit han önskar.

En menys logiska uppbyggnad skall i princip se ut så här:

1 * Dagliga rutiner	Rutiner som skall nås snabbt. Ex. Listning på skärm.
2 * Periodiska rutiner	Används periodiskt. Ex. Utskrifter.
3 * Registervård	Rutiner för speciella behov. Ex. Ändringar och tillägg.
4 * Systemvård	Används i princip bara en gång. Ex. Angivande av företagsnamn.
0 * Avslutning	Skall alltid stå sist i menyn. Innebär att man kommer till närmast föregående meny.

Skall användaren lämna en meny skall det bara vara möjligt att komma till menyn närmast ovanför eller nedanför i hierarkin. Man skall samtidigt undvika att göra CHAIN till rutiner som används ofta.

Vad man framför allt bör tänka på är att alla program inom ett system skall se ut på samma sätt.

För övrig information om hur programmen bör vara uppbyggda hänvisas läsaren till Metodhandboken utgiven av LUXOR och FACIT (ref. 3).

2.2.2 Strukturerad programmering med JSP

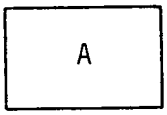
JSP är en metod för konstruktion av strukturerade program. Vad som kommer att behandlas på följande sidor är bara grunderna i JSP. Intresserade läsare kan för ytterligare information läsa boken, JSP - En praktisk metod för programkonstruktion (ref. 4).

Fördelarna med strukturerad programmering är att läsbarheten ökar, det blir lättare att ändra i programmet efteråt och det är lättare för andra att sätta sig in i programmet. Alla "vilda" GOTO, som snabbt leder till "spaghetti-syndromet", förvinner också.

Teoretiskt består alla program av tre (3) komponenter, nämligen:

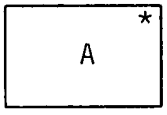
* Sekvens Utför instruktionerna sekvensiellt.

Beteckning:



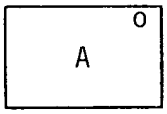
* Iteration Repetition. Ex. WHILE-, FOR-sats.

Beteckning:



* Selektion Antingen eller. Ex. IF-sats.

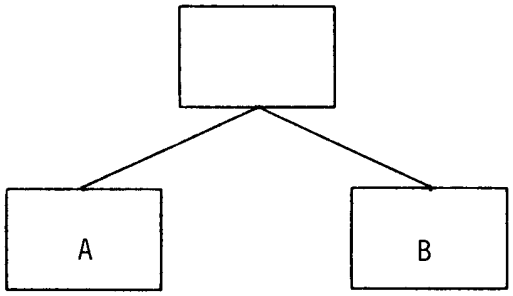
Beteckning:



Komponenterna ser ut på följande sätt i JSP- respektive BASIC-notation:

* Sekvens

JSP-notation:

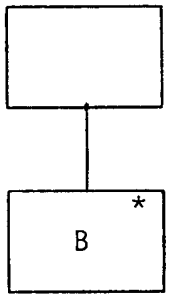


BASIC-notation:

```
10000 Z=FNBoxa ! Box A
10010 Z=FNBoxb ! Box B
```

* Iteration

JSP-notation:

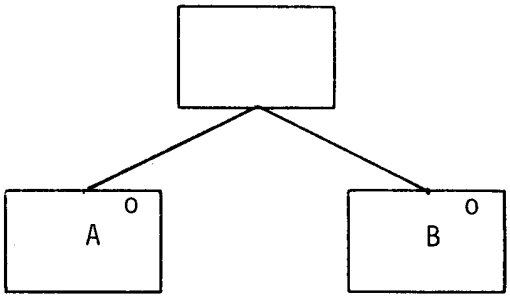


BASIC-notation:

```
10000 WHILE NOT Villkor
10010 Z=FNBoxb
10020 WEND
```

* Selektion

JSP-notation:



BASIC-notation:

```
10000 IF NOT Villkor THEN Z=FNBoxb ELSE Z=FNBoxa
10010 ! ENDIF
```

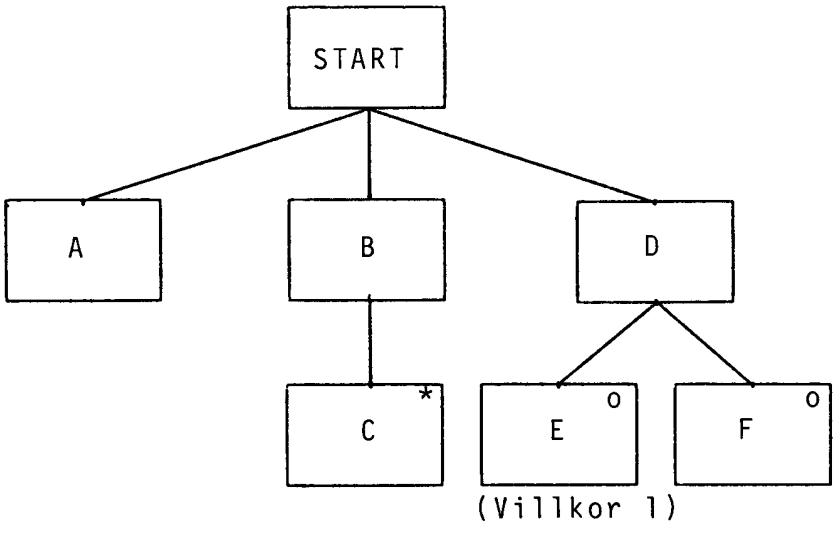
Det är bevisbart att alla program kan skrivas med endast dessa tre komponenter.

Observera att varje enstaka ruta ej ger besked om vilken komponenttyp den tillhör utan man måste alltid titta på närmast underliggande nivå. Varje box måste tillhöra en av de tre komponenterna entydigt, dvs man får ej ansluta en sekvens och en iterationsbox eller en iterations- och en selektionsbox till samma ruta.

Samtidigt skall programmet ha exakt en (1) ingång och en (1) utgång. Man kan alltså inte som i en flödesplan skriva in START och STOPP-rutor var som helst.

En olikhet mot programmeringsspråken är hur tidsföljden markeras. I JSP används horisontell markering, och man bygger upp programmet som en trädstruktur, medan man i vanliga programmeringsspråk använder en vertikal tidsföljdsmarkering.

Ett exempel får visa:



```
10000 ! Start
10005 !
10010 Z=FNBoxa ! Box A
10020 ! Start iteration Box B
10030 WHILE NOT Villkor
10040   Z=FNBoxc ! Box C
10050 WEND
10060 ! Slut iteration
10070 ! Start selektion Box D
10080 IF NOT Villkor1 THEN Z=FNBoxf ELSE Z=FNBoxe
10090 !ENDIF
10100 ! Slut selektion
10110 ! Stopp
```

Vi ser i exemplet ovan att JSP leder till blockstruktur på programmen, vilket ökar läsbarheten.

2.2.3 Generell programstruktur

En generell programstruktur för ett ABC800-program bör se ut på följande sätt:

1 * Deklarationer	Här deklarerar du dina COMMON variabler och dimensionerar variablerna med DIM-satser.
2 * Konstanter	Här placerar du dina variabler som skall ha ett konstant värde genom hela programmet. Dessa variabler får ej tilldelas mer än en (1) gång.
3 * Funktioner	Här finns alla dina funktions-(procedur-) definitioner.
4 * Huvudprogrammet	Här placeras "huvudloopen".

Varför vi har valt denna programstruktur beror på att ABC800 BASIC alltmer har närmat sig procedurorienterade språk, typ PASCAL. Dessa språk har som standard att först skall deklarationer och deklarering av konstanter komma och efter dessa följer procedurerna och funktionerna och sist följer huvudprogrammet.

Vad som har gjort att ABC800 BASIC har närmat sig dessa språk är införandet av flerradiga funktioner.

I och med att man använder funktioner får man följande effekter:

* En noggrann indelning i lokala och globala variabler.

* En modulindelning av programmet där varje logisk del är noggrant definierad.

* En blockstruktur som innebär att när ett program är skrivet så har man de flesta av de funktioner som behövs i nästa program. Exempel på sådana rutiner är: Filhanteringsrutiner
  Menyhanteringsrutiner

Programmerare som har använt tex PASCAL vet att i sådana språk finns procedur-begreppet (PROCEDURE) som tillägg till funktionsbegreppet. Skillnaden mellan dessa är att en funktion alltid returnerar ett värde, medan en procedur aldrig returnerar ett värde utan bara utför instruktionerna som finns i proceduren.

Procedur-begreppet går att införa i BASIC genom att man låter sin funktion returnera ett oväsentligt värde. Typiska värden är:

```
0  dvs RETURN 0      Vid INTEGER-funktion FNX(Y)
0. dvs RETURN 0.     Vid FLOAT-funktion   FNX.(Y)
""  dvs RETURN ""    Vid STRING-funktion  FNX¤(Y)
```

Exempel:

```
10 DEF FNProcedur
20   ! Utför någonting
30   ! fortsätt
40   ! proceduren klar
50 RETURN 0 ! Återvänd från proceduren
60 FNEND
```

Anrop av flera sekvensiella "procedurer" kan med hjälp av flerradiga strängfunktioner se ut på följande sätt:

```
1000 PRINT FNProca¤;
1010 PRINT FNProcb¤;
1020 PRINT FNProcc¤;
```

Funktionen (proceduren) utför bara instruktionerna som finns mellan DEF FN och FNEND. Den returnerar visserligen ett värde, men det är ett dummy-värde, dvs ett värde som det anropande programmet aldrig kommer att använda.

Användning av flerradiga funktioner underlättar inte bara läsbarheten utan även provkörning av olika programavsnitt.

Test av funktionen görs på följande sätt:

1 Ladda in funktionen i minnet.
2 Skriv RUN för att "FIXUP"-a programmet.
3 Skriv tex PRINT FNFunktion(Testvärde).

Funktionen kommer nu att genomköras och returnera funktionsvärdet, som kommer att skrivas ut på bildskärmen.

På detta sätt kan de flesta delarna i ett program felsökas var för sig.

2.2.4 Tips och fällor

Vid programmering med BASIC II finns det en del saker att tänka på. Några av dessa finns beskrivna här nedan. Framförallt effekterna av den nya ON ERROR-behandlingen.

LÅNGA VARIABELNAMN

Vid borttagning av programdelar (tex med ERASE) tas ej berörda långa variabelnamn bort ur listan med långa variabelnamn. För att spara plats - LISTA programmet och ladda in det igen med jämna mellanrum.

HASTIGHETSOPTIMERING

Undvik REM i loopar. Varje REM tar ca 200 mikrosekunder att utföra.

WHILE tar längre tid än FOR. Ändå bör FOR-loopar undvikas i funktioner pga att loopvariabeln INTE kan vara lokal. FOR-loopen är specialoptimerad för att kunna exekveras snabbt.

I de fall då heltalsvariabler kan användas bör man göra det eftersom dessa medger en snabbare exekveringstid.

Om en loop uförs många gånger - kontrollera om konstanter beräknas inuti loopen. I så fall flytta dem utanför loopen.

Exempel:

```
0000 ! Vridning av tvådimensionella koordinater
0005 !
0010 FOR I=0 TO 10000
0020   X1.(I)=COS(A.)*X0.(I)-SIN(A.)*Y0.(I)
0030   Y1.(I)=COS(A.)*Y0.(I)+SIN(A.)*X0.(I)
0040 NEXT I
```

Byt ut följande rader:

```
0005 Xr.=COS(A.): Yr.=SIN(A.)
0020 X1.(I)=Xr.*X0.(I)-Yr.*Y0.(I)
0030 Y1.(I)=Xr.*Y0.(I)+Yr.*X0.(I)
```

Detta kommer att snabba upp programmet avsevärt.

PLATSOPTIMERING

Packning av utskriftselement med semikolon (;) tar en byte mer per semikolon än motsvarande packning med space.

Ex:
PRINT A¤;B¤;C¤ tar 2 byte mer än
PRINT A¤ B¤ C¤

Negativa INTEGER konstanter tar 1 byte mer än motsvarande positiva.

Exempel:
POKE -767,0 tar 1 byte mer än
POKE 64769,0

Konstanter i området 0 - 16 lagras i 1 byte.

Konstanter i området -0 - (-16) lagras i 2 byte.

Flyttalskonstanter som är "binära", dvs talet är lämpligt att beskriva på binärform, tar mindre plats än motsvarande icke "binära" flyttalskonstanter.

Exempel:
2. tar mindre plats än 2.37

Skrivsättet
10 xxxxxx : ! ABC   tar 1 byte mer än
10 xxxxxx ! ABC

Undvik att ha kolon med när en REM-sats följer direkt efter en exekverbar sats.

HUR MYCKET PLATS TAR VARIABLERNA ?

Olika variabeltyper tar olika plats. Dessutom tar det olika lång tid att referera till olika variabeltyper.

Skalära INTEGER/FLOAT-variabler tar mer lagringsplats än respektive vektorer/matriser. Däremot refereras skalära INTEGER/FLOAT-variabler snabbare.

Skalära strängvariabler tar både mindre plats och refereras snabbare än motsvarande vektorer/matriser.

Här följer en uppsättning platsberäkningsformler:

	Skalär	Vektor/matris
HELTAL	6	10 + 6 * ia + 2 * ea
FLYTTAL I SINGLE	8	10 + 6 * ia + 4 * ea
FLYTTAL I DOUBLE	12	10 + 6 * ia + 8 * ea
STRÄNG	10+diml	10 + 6 * ia + 6 * ea + diml * ea

ia   = index-antal
ea   = element-antal
diml = dimensionerad längd

Exempel:

En flyttalsvektor (enkel precision) med 11 element upptar
10 + 6 * 1 + 4 * 11 = 60 byte.

En strängmatris med 11*11 element och längden 80 tecken upptar
10 + 6 * 2 + 6 * 121 + 80 * 121 = 10428 byte.

FLERRADIGA FUNKTIONER, PROBLEM

Vid användning av flerradiga funktioner tillsammans med globala variabler, som tilldelas inuti funktionen, kan (ibland) vissa situationer uppstå.

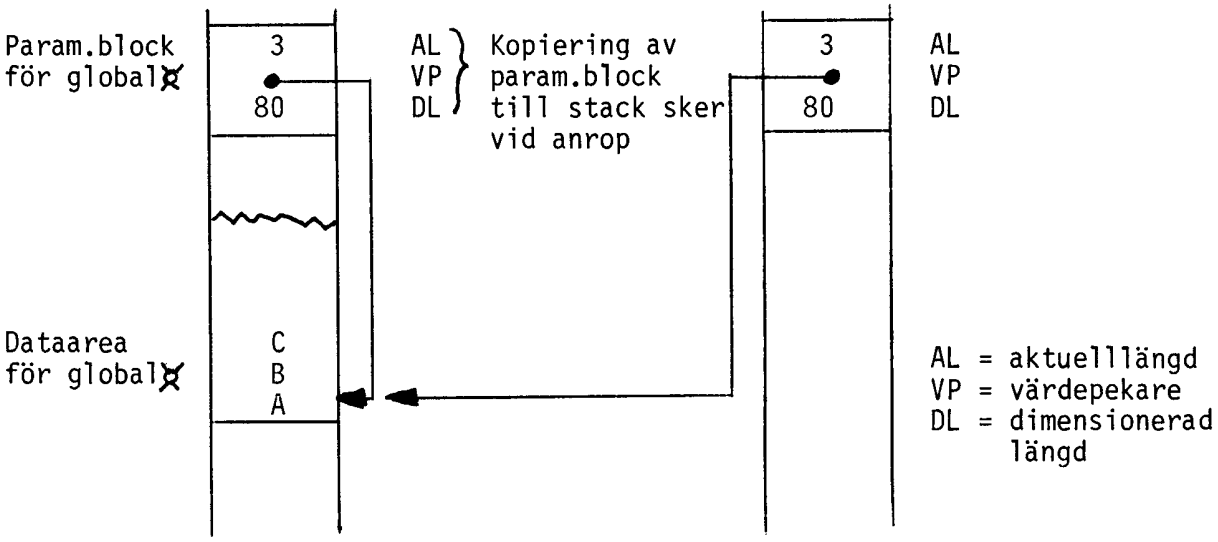
Exempel 1:

Vi använder den globala variabeln Global¤ som inparameter. Tilldelar Global¤ inuti funktionen och returnerar inparametern. (Inte speciellt vanligt men i alla fall.)

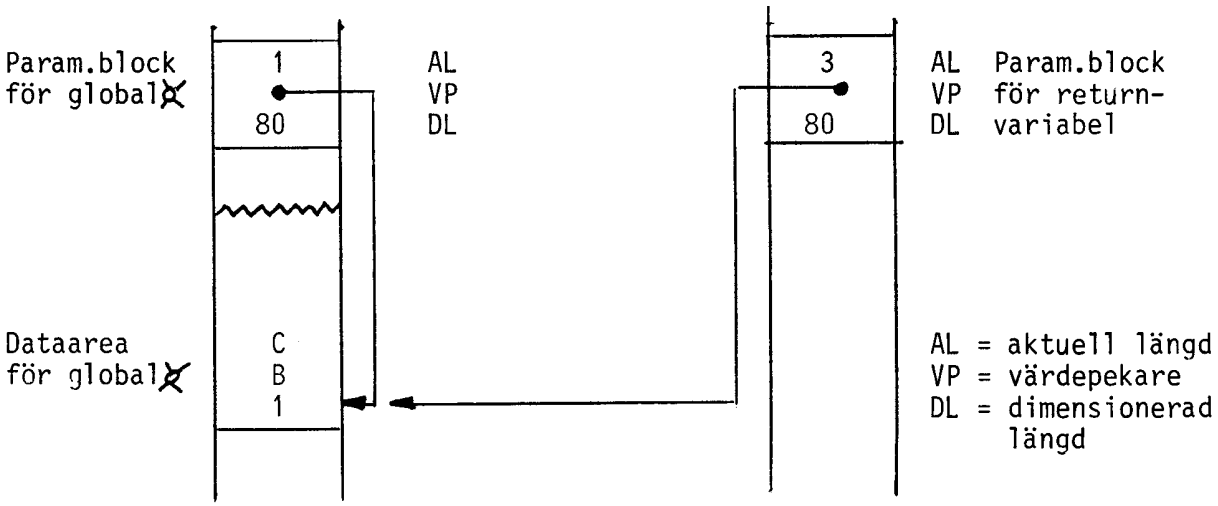
```
10000 DEF FNGlobal¤(In¤)
10020  Global¤="1"
10030  RETURN In¤
10040 FNEND
10050 Global¤="ABC"
10060 ; FNGlobal¤(Global¤)
```

Detta exempel ger resultatet 1BC. INTE ABC eller 1.

Vid anropet kopieras parameterblocket för strängen Global¤ till arean för funktionens inparametrar. Parameterblocket (det som ligger som inparameter) pekar dock på samma dataarea som originalblocket. När vi nu förändrar Global¤ inuti funktionen på rad 10020, förändras aktuell längd i det ursprungliga parameterblocket för Global¤, som får aktuell längd 1. När vi sedan returnerar inparametern (som innehåller aktuell längd 3) kommer felaktiga data att returneras.



Utseende vid funktionsanrop



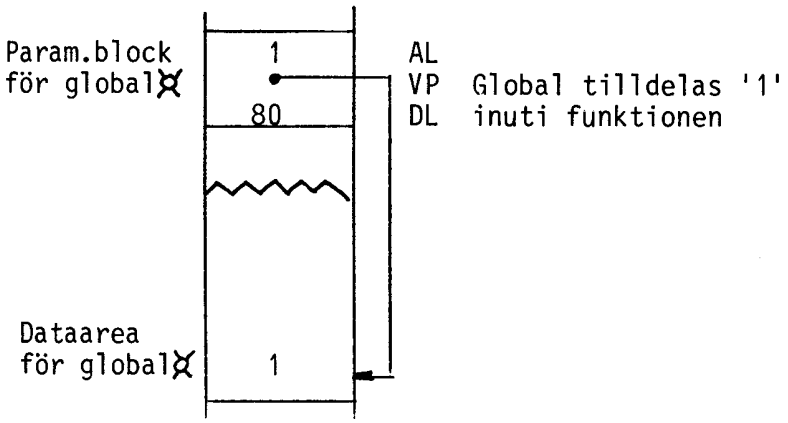
Utseende vid RETURN från funktion

Exempel 2:
Vi använder den globala variabeln Global¤ inuti funktionen, som returnvariabel, och anropar funktionen flera gånger med olika inparametrar i en konkatenering.

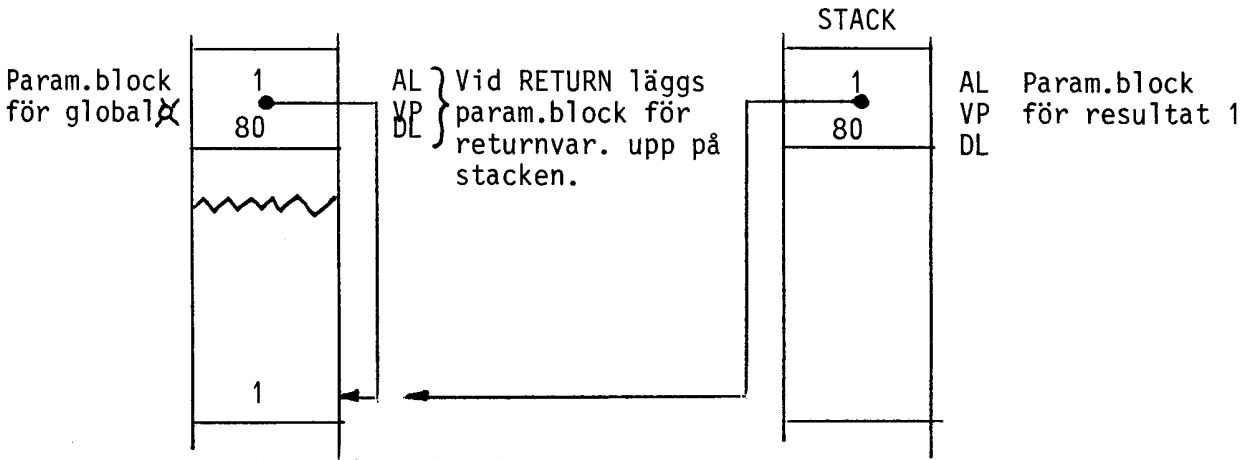
```
10000 DEF FNN¤(X)
10020   Global¤=NUM¤(X)
10030   RETURN Global¤
10040 FNEND
10050 ; FNN¤(1)+FNN¤(2)
```

Detta exempel ger resultatet 22. INTE 12.

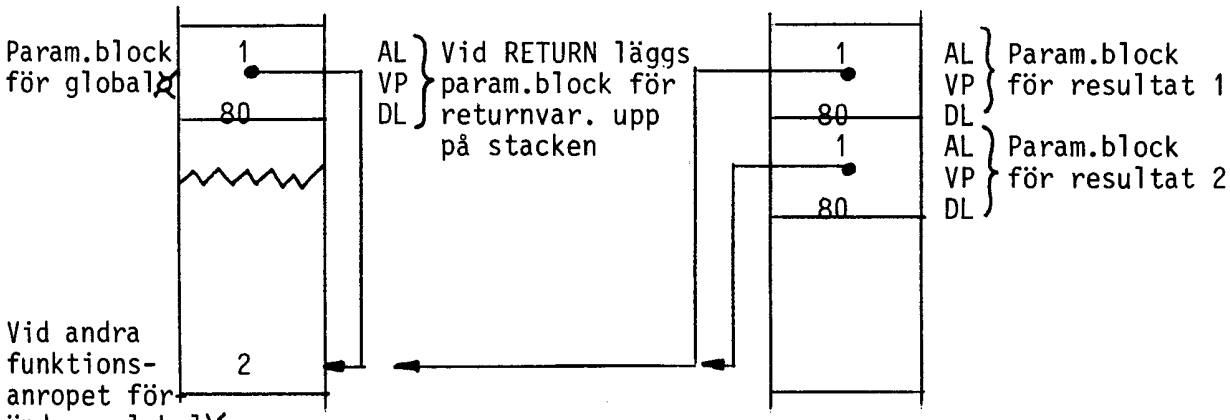
Vid första anropet av FNN¤, kopieras parameterblocket för Global¤ till stacken, som håller mellanresultatet. Vid andra anropet kommer rad 10020 att "skriva över" resultatet från första anropet. Båda kopiorna av parameterblocken pekar på samma dataarea, som nu innehåller 2, och när konkateneringen utförs blir resultatet 22.



Utseende vid första RETURN



Utseende vid andra RETURN



AL = Aktuell längd
VP = Värdepekare
DL = Dimensionerad längd

FLERRADIGA FUNKTIONER, REKURSION

Flerradiga funktioner tillåter rekursion, dvs funktionen kan anropa sig själv ett upprepat antal gånger. Vid varje anrop av en funktion läggs inparametrar och lokala variabler upp i en ny dataarea. Detta medför att rekursion är platskrävande ifråga om variabelutrymme (beroende på det antal rekursioner som utförs) men oftast platsbesparande ifråga om kodutrymme.

Många problem typ sökningar och upprepade beräkningar löses ofta med hjälp av rekursion (se FNBinsök i bilaga 2 och nedanstående exempel). I exemplet är en fakultetsberäkning utförd med hjälp av rekursion. Funktionen finns i exemplet dels som flyttalsfunktion och dels som strängfunktion (medför att upp till 33! respektive 82! kan beräknas).

Exempel:

```
10000 DEF FNFak.(N.)
10010   IF N.<2. THEN RETURN 1. ELSE RETURN N.*FNFak.(N.-1.)
10020 FNEND
10030 !
10040 ;FNFak.(33.)

10000 DEF FNFak¤(N¤)
10010   IF COMP%(N¤,"1")<1 THEN RETURN "1"
        ELSE RETURN MUL¤(N¤,FNFak¤(SUB¤(N¤,"1",0)),0)
10020 FNEND
10030 !
10040 ;FNFak¤("82")
```

ERRORHANTERING

Errorhantering i samband med flerradiga funktioner kan bli lite knepig i och med att ett ON ERROR GOTO kan användas inuti funktionen samtidigt som ON ERROR GOTO är satt utanför funktionen.

ON ERROR GOTO-behandling utan RESUME

Ett ON ERROR GOTO som sätts inuti en funktion benäms härefter som ett lokalt ON ERROR GOTO och ett ON ERROR GOTO som sätts utanför funktionen som globalt.

När man använder ON ERROR GOTO utan RESUME gäller följande regler:

* Ett globalt ON ERROR GOTO gäller i funktionen tills dess att ett ON ERROR GOTO sätts inuti funktionen.

* Om fel kan inträffa i funktionen måste lokalt ON ERROR GOTO användas.

Exempel:

```
10 DEF FNA.(B.)
20   RETURN SIN(B.)/COS(B.) ! Om fel inträffar här hamnar vi på
                              rad 100 vilket inte är lämpligt.
30 FNEND
35 !***************
40 ON ERROR GOTO 100
50 INPUT A.
60 ; FNA.(A.)
70 ....
80 ....
90 ....
100 ! FELRUTIN
110 ....
```

* Ett ON ERROR GOTO <rad> som sätts i en funktion tas bort när RETURN exekveras. Satsen ON ERROR GOTO behöver ej användas.

Exempel:

```
10 DEF FNA.(B.)
20   ON ERROR GOTO 40
30   RETURN SIN(B.)/COS(B.) ! Om fel inträffar här hamnar vi på
                              rad 40.
40   ! FELRUTIN
50   RETURN 999.
60 FNEND
65 !***************
70 ON ERROR GOTO 140
80 INPUT A.                 ! Inträffar fel här hamnar vi på rad
                              140.
90 ; FNA.(A.)               ! Nu gäller fortfarande ON ERROR GOTO
                              140.
100 ....
110 ....
120 ....
130 ....
140 ! FELRUTIN
150 ....
```

* När en funktion anropar en annan funktion gäller det senast aktuella ON ERROR GOTO tills dess att ett ON ERROR GOTO sätts i den anropade funktionen. När exekveringen återvänder till den anropande funktionen gäller det ON ERROR GOTO som fanns innan anropet skedde.

Exempel:

```
10 DEF FNA.(B.) LOCAL F.
20   ON ERROR GOTO 50
30   F.=FNB.(B.)
40   RETURN SIN(B.)/COS(F.) ! Om fel inträffar här hamnar
                              vi på rad 50.
50   ! FELRUTIN
60   RETURN 999.
70 FNEND
80 DEF FNB.(S.)
85   Z=CALL...              ! Inträffar fel här hamnar vi på
                              rad 50.
90   ON ERROR GOTO 110
100   RETURN COS(S.)/TAN(S.) ! Om fel inträffar här
                               hamnar vi på rad 110.
110   ! FELRUTIN
120   RETURN 999.
130 FNEND
135 !***************
140 ON ERROR GOTO 210
150 INPUT A.                 ! Inträffar fel här hamnar vi på
                               rad 210.
160 ; FNA.(A.)
170 ....                     ! Nu gäller fortfarande ON ERROR
                               GOTO 210.
180 ....
190 ....
200 ....
210 ! FELRUTIN
220 ....
```

ON ERROR GOTO-behandling med RESUME

OBSERVERA! Vid förekomst av instruktionen RESUME (någonstans i programmet) krävs att alla ON ERROR-hanteringar genomlöper RESUME för att återställa BASIC:ens stackhantering.

Exempel:

```
100 ON ERROR GOTO 140
110 INPUT A.                 ! Inträffar fel här hamnar vi på
                               rad 140.
120 ; FNA.(A.)
130 END                      ! Nu gäller fortfarande ON ERROR
135 !                          GOTO 140.
140 ! FELRUTIN
150 ; ´FEL´
160 RESUME                   ! I och med detta RESUME måste alla
                              felhanteringar "avslutas" med RESUME
165 !
170 DEF FNA.(A.)
180   ON ERROR GOTO 200
190   RETURN COS(A.)/TAN(A.) ! Här gäller ON ERROR GOTO 200.
200   ! FELRUTIN
210   RESUME 220              ! Om denna rad saknas kommer fel-
                               meddelande att genereras vid rad
                               220.
220   RETURN 999.
230 FNEND
```

2.3 Datahantering på sekundärminne

2.3.1 Sekvensiella filer

I en sekvensiell fil ligger posterna lagrade med CR, CHR¤(13), som avgränsare.

Sist i varje sektor ligger ETX, CHR¤(3), som innebär att man skall läsa in en ny sektor.

Filslut markeras med 6 st NUL, CHR¤(0).

Blanktecken (´ ´) komprimeras på formen CHR¤(9)+CHR¤(Antalet blanka).

Behandlar man posterna med hjälp av INPUT, PRINT och INPUT LINE, vilket rekommenderas, behöver man inte bry sig om ovanstående då systemet sköter om att man får rätt post. Ibland kan man dock vilja behandla en sekvensiell fil med "direkt access" och då kan det vara bra att känna till lite om lagringssättet. När man läser/skriver på en sekvensiell fil kan man bara komma åt posten efter den man läste/skrev sist.

Här följer nu ett litet program som läser in en fil,skriver ut dess poster och lagrar posterna i en ny fil.

```
10010 INTEGER : EXTEND
10020 OPEN ´TEST1.DAT´ AS FILE 1
10030 PREPARE ´TEST2.DAT´ AS FILE 2
10040 ON ERROR GOTO 10110 ! Behandla filslut på infilen
10050 WHILE 1 ! Oändlig loop. Terminering sker mha ON ERROR GOTO
10060 INPUT LINE #1,A¤
10070 ! Tag bort CHR¤(13,10) ur strängen
10080 PRINT A¤;
10090 PRINT #2,A¤;
10100 WEND
10110 CLOSE 1,2
10120 END
```

2.3.2 Random Access-filer

Hanteringen av random access-filer skiljer sig markant från hanteringen av sekvensiella filer. Här tänker man sig hela filen som en enda stor sträng, och en post är då ett antal av dessa tecken. Därför har man en fast postlängd när man hanterar random access-filer.

En diskett är indelad i spår och varje spår innehåller ett antal sektorer. Varje sektor är 256 bytes lång men av dessa är 3 byte information för DOS. Det återstår således 253 byte för användaren.

De instruktioner som finns för hantering av random access-filer är:

POSIT #filnr, position	Ställer filpekaren (pekare till var nästa läsning/skrivning sker på filen) till position "position" på filen med filnumret "filnr".
GET #filnr,variabel¤ COUNT längd	Läser "längd" antal tecken från filpekaren i filen "filnr" och lagrar dessa i variabeln "variabel¤".
PUT #filnr, variabel¤	Skriver ut variabeln "variabel¤" till filen "filnr" från filpekaren.

Fördelarna med random access är att man ej är begränsad av att behandla nästa post utan man kan direkt läsa/skriva vilken post som helst i filen. Random access går också fortare än sekvensiell behandling.

Här följer nu ett program som läser in en random access-fil och kopierar de sex första posterna med postlängden 80 till en ny random access-fil.

```
10000 INTEGER : EXTEND
10010 OPEN "TEST1.DAT" AS FILE 1
10020 PREPARE "TEST2.DAT" AS FILE 2
10030 Postlängd=80
10040 FOR I=0 TO 5
10050   ! LÄS IN POST
10060   GET #1,Poststräng¤ COUNT Postlängd
10070   ! SKRIV UT POST
10080   PUT #2,Poststräng¤
10090 NEXT I
10100 CLOSE 1,2
10110 END
```

Ovanstående exempel kan sägas vara en sekvensiell hantering av en random access-fil. Oftast förekommande är dock behovet av att kunna läsa och skriva valfri post.

Nedan finns exempel på rutiner för att läsa och skriva i en random access-fil med valfri postlängd.

```
10000 Fil=1
10010 OPEN "TEST1.DAT" AS FILE Fil
10020 Postlängd=97
10030 DIM Post¤=Postlängd
10040 DEF FNLäspost¤(Post,Fil) LOCAL Buf¤=256
10050   Ec=0
10060   ON ERROR GOTO 10100
10070   POSIT #Fil,Post*Postlängd
10080   GET #Fil, Buf¤ COUNT Postlängd
10090   RETURN Buf¤
10100   Ec=ERRCODE
10110   RETURN SPACE¤(Postlängd)
10120 FNEND
10130 !
10140 DEF FNSkrivpost(Post,Fil,Buf¤)
10150   Ec=0
10160   ON ERROR GOTO 10200
10170   POSIT #Fil,Post*Postlängd
10180   PUT #Fil, Buf¤
10190   RETURN 0
10200   Ec=ERRCODE
10210   RETURN Ec
10220 FNEND
```

När man gör PREPARE på en fil, reserveras ett större antal sektorer. Vid CLOSE återlämnas de sektorer som inte har utnyttjats. Ett exempel får visa.

Om vi antar att PREPARE reserverar 32 sektorer och programmet har använt 6 sektorer så återlämnas 26. Skriver vi nu på 10:e sektorn i filen reserverar ABC800 plats för sektorerna 7-9. Ett problem här är om programmet tex skall läsa den 7:e sektorn utan att ha skrivit på den förut. I detta fall genereras error 37. Som exempel kan vi ta programmet nedan som ger utskriften:

```
SKAPAT FILEN
LAGT TILL POST
Error 37 in line 10150
```

```
10010 INTEGER : EXTEND
10020 PREPARE "TEST.DAT" AS FILE 1
10030 FOR I=0 TO 6
10040   A¤=STRING¤(253,65+I)
10050   PUT #1,A¤
10060 NEXT I
10070 CLOSE 1
10080 ;"SKAPAT FILEN"
10090 OPEN "TEST.DAT" AS FILE 1
10100 POSIT #1,20*253
10110 A¤=STRING¤(253,65+20)
10120 PUT #1,A¤
10130 ;"LAGT TILL POST"
10140 POSIT #1,10*253
10150 GET #1,A¤ COUNT 253
10160 ;A¤
10170 ;"LÄST POST"
10180 CLOSE 1
10190 END
```

2.3.3 Adresseringsmetoder

Vi skall här gå igenom olika metoder för att hitta en post på en fil med random access-struktur.

INDEXERAD ADRESSERING

Indexerad adressering används när man vill underlätta en sekvensiell bearbetning av posterna. Lagring av posternas id-värden sker då, tillsammans med en adressangivelse till postens faktiska position, i en s.k indextabell. För att snabbt kunna upptäcka om en post inte finns, utan att behöva söka igenom hela tabellen, sorteras vanligtvis denna.

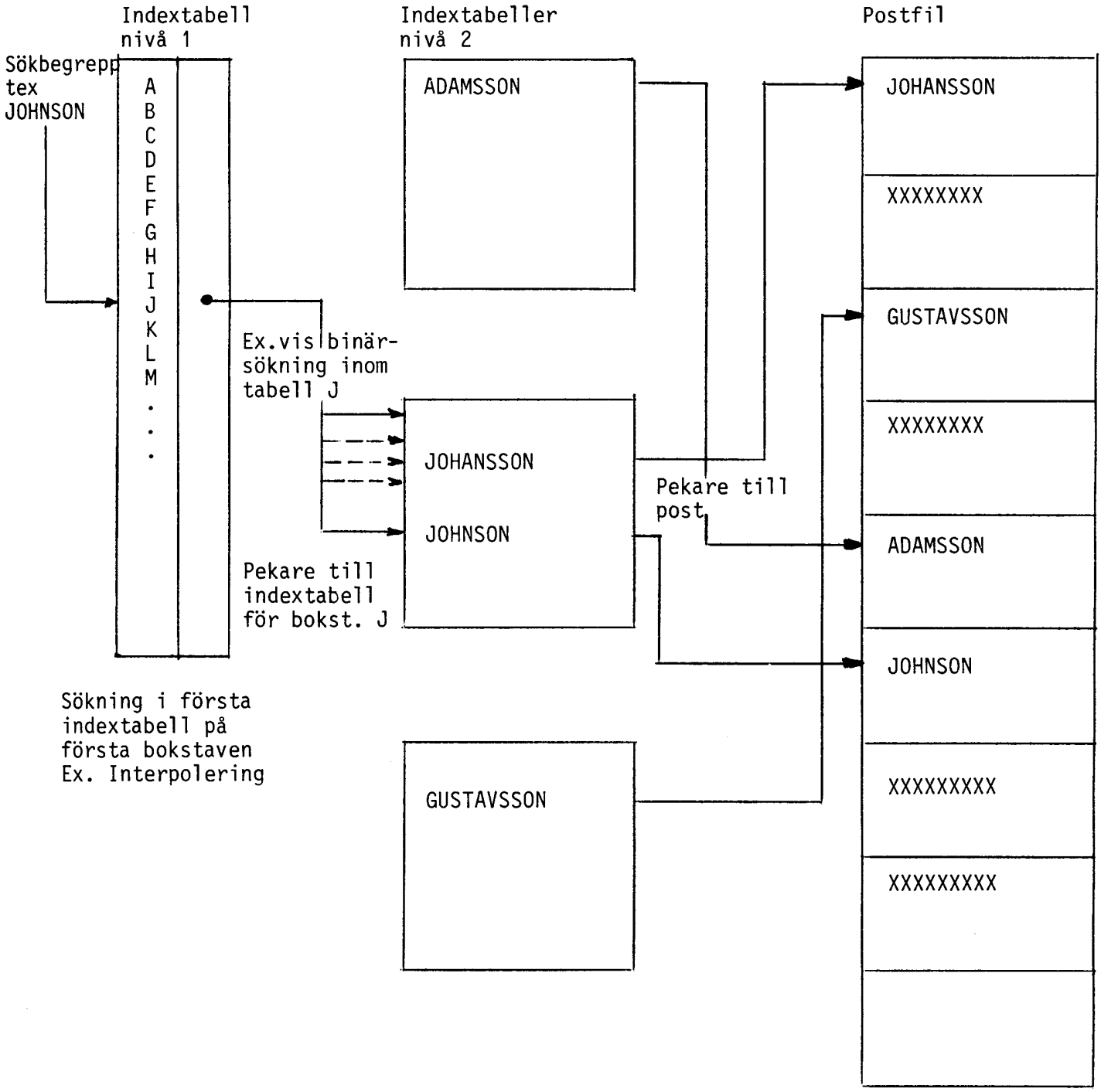
Vid hantering av större filer är det lämpligt att dela upp indextabellen i flera tabeller (i nivåer).

Sökningen går då till på följande sätt :

1 Sökning utförs i den första indextabellen.

2 Resultatet av första sökningen är en adressangivelse till en underordnad tabell.

3 Sök i den underordnade tabellen, osv tills postens faktiska adress är funnen.


Indextabell
nivå 1
Indextabeller
nivå 2
Postfil
Sökbegrepp
tex
JOHNSON
A
B
C
D
E
F
G
H
I
J
K
L
M
. . .
ADAMSSON
JOHANSSON
XXXXXXXX
GUSTAVSSON
XXXXXXXX
ADAMSSON
JOHNSON
XXXXXXXXX
XXXXXXXXX
Ex.vis binär-
sökning inom
tabell J
JOHANSSON
JOHNSON
Pekare till
post
Pekare till
indextabell
för bokst. J
Sökning i första
indextabell på
första bokstaven
Ex. Interpolering
GUSTAVSSON

SÖKGÅNG

SÖKNING I INDEXTABELLER

Metoder för sökning i indextabeller kan vara:

* Sekvensiell sökning
* Binärsökning
* Interpolering
* Pseudoadressering (se separat avsnitt)

Dessa sökmetoder kan med fördel även tillämpas vid sökning i andra tabeller.

SEKVENSIELL SÖKNING

Man jämför den sökta postens id-värde med id-värdena i tabellen från tabellens början tills man hittar posten eller tills tabellen är slut.

I medeltal behöver man då N/2 sökningar där N = antalet poster i tabellen.

BINÄRSÖKNING

Vid binärsökning sker ej sökning från tabellens början eller slut, som vid sekvensiell sökning, utan från tabellens mitt. Man avgör nu om sökt post, om den finns, är belägen i den övre eller undre delen av tabellen. I den del som posten kan finnas i upprepas ovanstående sökgång. På detta sätt har man minskat tabellstorleken med lite mer än hälften.

Ovanstående upprepas tills posten är funnen eller tills dess att fler delningar av tabellen ej är möjligt.

Maximalt antal sökningar som behövs med denna metod beräknas ur:

-INT(-LOG(N+1)/LOG(2))

där N = Antalet poster i tabellen.

Exempel:
För 1000 poster krävs maximalt 10 sökningar.

För exempel på hur en rekursiv binärsökning kan gå till refereras till funktionen FNBinsök i bilaga 2.

INTERPOLERING

Om id-värdena är någorlunda jämt fördelade kan man jämföra det högsta och lägsta id-värdet med det sökta och ungefärligt räkna ut var den sökta posten borde vara belägen. Efter en första jämförelse kan man sedan succesivt pejla in det sökta id-värdet, som vid binärsökning eller sekvensiell sökning.

PSEUDOADRESSERING (HASH-METOD)

Pseudoadressering används när identifikationsbegreppet för posterna är spridda över ett stort område.

Metoden går ut på att transformera id-begreppens värde till en acceptabel adress enligt en given algoritm (sk randomiseringsalgoritm). Den kan se ut på olika sätt men de vanligaste är "mittersta kvadrat-metoden" och "division rest-metoden".

"MITTERSTA KVADRAT-METODEN"

Denna metod går ut på att kvadrera id-värdet. (Om id-värdet ej är numeriskt kan man exempelvis addera ihop id-värdenas ASCII-värden. 'ABC' skulle då ge id-värdet 65+66+67=198.) Sedan använder man ett lämpligt antal siffror plockade ur mitten av talet.

För att förenkla uttagningen av de "mittersta" siffrorna kan talet omvandlas till en sträng med funktionen NUM¤( ).

Kan man inte få ut några siffror ur mitten (strängen blev för kort), fylls strängen ut med nollor före och efter tills det är möjligt.

Ofta kan det vara så att man tex bara har 500 poster men id-begreppet kan ha värden från 0 till 999 eftersom 500 medför 3 id-siffror. Man måste då krympa adressområdet ännu mer genom att multiplicera med en lämplig faktor (i det här fallet 500/1000 =0.5). Detta ger, efter att man har tagit INT av resultatet, ett talområde på 0-499.

DIVISION REST-METODEN

I stället för kvadrering divideras det numeriska id-värdet med antalet poster. Om id-värdet ej är numeriskt så kan man använda samma metod som beskrevs under "mittersta kvadrat-metoden".

Resten vid divisionen används som adress, dvs post med numeriskt id 1235 i ett register med max 1000 poster får adress:

ADRESS=ID-INT(ID/POSTANTAL)*POSTANTAL

Adress = 1235 - INT(1235/1000)*1000

På ABC800 kan detta enkelt beräknas med MODULO (MOD)-funktionen. Ovanstående blir då:

Adress = MOD(1235,1000) dvs 235.

Observera att man som "antal poster" bör välja ett primtal då det har visat sig att detta ger bättre spridning. Om man t.ex. har 1000 poster väljer man 997 eller 1003 till algoritmen. En variant på detta är att man vet totala antalet poster, tex 1000, och primtalet närmast under detta tal (997 för 1000). Då fås hemadresen ur:

Adress = INT(MOD(idvärde,primtal)*postantal/primtal)

ÖVRIGA METODER

Ibland är det inte nödvändigt att använda någon av ovanstående algoritmer utan det räcker med mycket enklare medel. Att dividera id-värdet med 10, eller att ta ASCII-koden för första tecknet i posten kan räcka för att få en acceptabel adress.

PROBLEM MED PSEUDOADRESSERING

Randomiseringsalgoritmen kan leda till att flera poster får samma adress (sk hemadress). Detta innebär att alla poster inte får plats på sin hemadress utan får läggas någon annanstans, sk overflow-areor. För att vi skall kunna hitta en post som ligger i overflow-arean placerar vi en pekare, dvs en adressangivelse, till postens adress i hemadressen. Innehåller overflow-arean flera poster med samma hemadress får varje sådan post innehålla en pekare till nästa post. Detta leder till att vi får en länkad lista.

Ett annat vanligt problem är att man får flera poster med samma id-värde. Om id-värdet är efternamn kan man ex.vis ha 100 st JOHANSSON.

HANTERING AV OVERFLOW-AREOR

Overflow-arean kan antingen ligga före eller efter området där hemadresserna finns,den sk primärarean. Detta är mycket minneskrävande.Man kan utnyttja att det uppstår luckor i primärarean och placera sina synonymer där (poster som ej fick plats på hemadressen).

Vid den senare tekniken är den enklaste metoden att hitta ledigt utrymme att söka sekvensiellt från hemadressen efter första lediga post. Om man kommer till slutet på arean börjar man om från början, sk cyklisk sökning. Ett annat sätt, för att få bättre spridning,är att använda randomfunktionen som när den är större än ett visst värde innebär att man söker nedåt i arean från hemadressen och annars söker uppåt.

Om man använder pseudoadressering på fil kan man nedbringa antalet accesser genom att samla flera poster, som har erhållit samma hemadress, i s.k. "buckets". Vid läsning/skrivning överförs hela "buckets" mellan primär- och sekundärminnet. Får posterna inte plats i sin "bucket" kan man antingen öka "bucket"-storleken om man har minne nog, eller använda någon av ovanstående metoder.

UPPDATERING AV POSTER

Vid inmatning av nya poster, som skall lagras, kontrollerar man om det finns plats på hemadressen. Om det finns plats lagras posten där. Annars lagras posten i overflow-arean. Vid radering av poster kan det hända att en hemadress för en post blir ledig. Då kan man flytta en post från overflow-arean till hemadressen. Men vid stora poster kan detta bli otympligt. I dessa fall brukar man bara flytta pekare och märka hemadressen som ledig och låta pekaren till overflow-arean ligga orörd.

Om man vid borttagning av poster i overflow-arean vill använda samma handhavande och vid speciella tillfällen "städa" listan, eller om man direkt vill flytta pekarna är en smaksak.

Den som vill läsa mer om olika adresseringsmetoder hänvisas till boken "Datasystem och Datorsystem", Sam Nachmens (Ref. 5).

2.3.4 ISAM

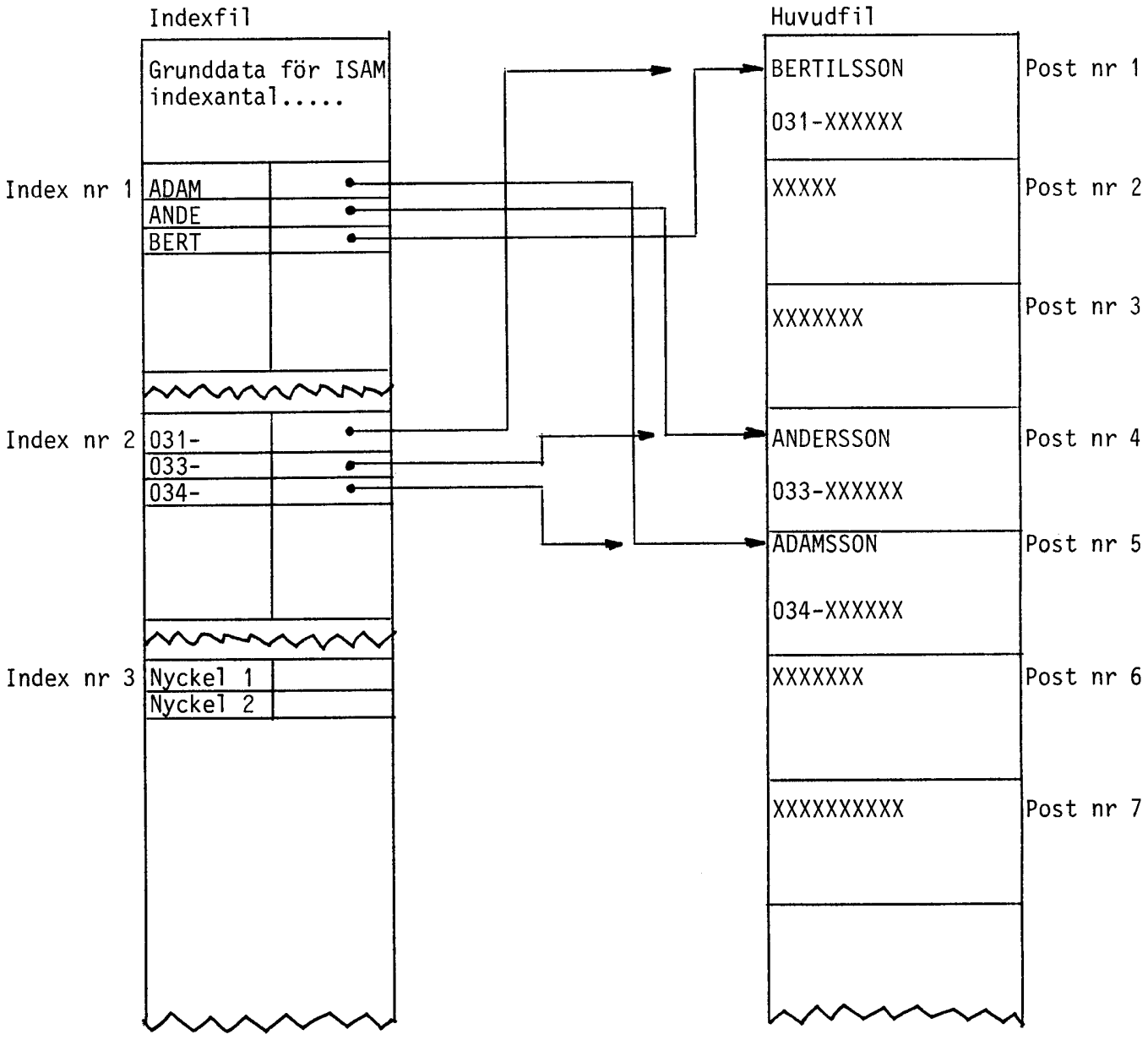
ISAM (Index-Sekvensiell-Access-Metod) är en metod att organisera ett register för att snabbt kunna söka poster. Posterna är alltid sorterade. Sortering sker vid inmatning.

ISAM på ABC800

På ABC800 finns ISAM att tillgå som ett hjälppaket med ett antal nya BASIC-instruktioner. ABC800-ISAM laddas från diskett.

ISAM medför att flera program i en ABC800 Multi-User miljö kan arbeta mot samma fil.

ISAM arbetar med två filer. En fil innehåller alla poster med komplett information. Fil nr 2 innehåller bara de sökbegrepp, sk nycklar (med adresspekare till fil nr 1), som är definierade.



Vid sökning av en post läses först nyckelfilen för att få adress-pekaren till aktuell post. Därefter läses posten i huvudfilen. För att sökning skall gå snabbt är nyckelfilen alltid sorterad.

Antalet nycklar till en post kan variera från 1 till 10, dock bör man inte överdriva antalet nycklar då dessa dubbellagras. I en ISAM-fil (där alla fält också är nycklar) tar varje post upp mer än dubbelt så mycket plats som vid sekvensiell lagring.

DEFINIERA ISAM-fil

Indexfilen har nedanstående utseende. Antingen kan den skapas med hjälpprogrammet CREINDEX.BAC, som följer med ISAM-paketet, eller också kan den skapas med ett eget program.

Pos	ASCII	Förklaring
1	1	ISAM typ.
2	0-1 och 8-9 eller 255	Drivenummer för postfil. Om drivenr=255 letas filen upp.
3-10		Namn på postfil, 8 pos.
11-13		Typ på postfil, 3 pos.
14	3	
15-16		Antal index.
17-25	0	För senare bruk.
26-27		Postlängd.
28-35		Indexnamn, 8 pos.
36	3	
37-38		Indexnummer.
39		Duplicerade nycklar J/N.
40-41		Startpos i post för index.
42		Längd på index.
43		Typ på index.
44-47	0	För senare bruk.
48-65		Pos 28-47 upprepade för index nr 2.
66-83		Pos 28-47 upprepade för index nr 3.
.		
.		
..-..		Pos 28-47 upprepade för index nr n.

Duplicerade nycklar är: Ja = 1 (binärt)
Nej = 0 -"-

Typ är: Binary = 0 (binärt)
ASCII = 1 -"-
Integer = 2 -"-
Float = 3 -"-
Double = 4 -"-

Efter alla indexupprepningar skall 10*253 bytes med CHR¤(0) följa.

Postfilen måste vara skapad (med PREPARE) men skall inte innehålla någon information.

NYA BASIC-instruktioner för ISAM

ISAM OPEN - Öppnar ISAM-filen för bearbetning.

ISAM WRITE - Skriver post i filen.

ISAM READ - Läser post i filen.

ISAM UPDATE - Uppdaterar befintlig post.

ISAM DELETE - Tar bort befintlig post.

ISAM OPEN

Innan bearbetning av en ISAM-fil kan påbörjas måste den öppnas som alla andra filer. Den vanliga OPEN-instruktionen kan ej användas utan istället används följande instruktion:

100 ISAM OPEN ´<enhet:>filnamn<.typ>´ AS FILE nr

Syntaxen är identisk med syntaxen för den vanliga OPEN-instruktionen. Standard typ är ".ISM"

OBSERVERA att namnet, som anges, skall ange INDEX-filen, inte postfilen.

En ISAM-fil stängs med den vanliga CLOSE-instruktionen.

ISAM WRITE

Skriver en ny post i ISAM-filen.

100 ISAM WRITE #nr,sträng

"Sträng" skall innehålla hela posten med fälten utfyllda till maximal längd.

ISAM READ

Används när filen skall läsas i en sorterad ordning eller vid sökning av enskild post.

```
100 ISAM READ #nr,sträng1 <INDEX sträng2> <KEY sträng3>
                                          <FIRST>
                                          <LAST>
                                          <NEXT>
                                          <PREVIOUS>
```

Resultatet av inläsningen läggs i sträng1 på samma sätt som vid INPUT.

Läsningen sker efter det index som namnges av INDEX sträng2. Anges inget index kommer sist använda eller, om ingen tidigare läsning skett, första index att användas.

Om sökning på specifik nyckel önskas anges det med KEY sträng3, där sträng3 innehåller önskad nyckel.

För utskrifter i sorterad ordning anges i stället för KEY orden FIRST, LAST, NEXT eller PREVIOUS som läser första, sista, nästa respektive föregående post.

ISAM UPDATE

Uppdaterar en befintlig post. Alla nycklar uppdateras.

100 ISAM UPDATE #nr,Sträng1 TO Sträng2

Sträng1 måste innehålla resultatet från föregående läsning. Sträng2 innehåller den nya strängen som skall skrivas istället för Sträng1.

OBSERVERA att ISAM UPDATE måste föregås av ett ISAM READ. Om medskickad "Sträng1" ej överensstämmer med posten från föregående läsning ges ERROR 123. Detta används för att kontrollera uppdateringen i ett Multi User-system.

ISAM DELETE

Tar bort en post ur ISAM-filen.

100 ISAM DELETE #nr,Sträng

"Sträng" måste innehålla resultatet från föregående läsning.

OBSERVERA att den post som skall tas bort måste före ISAM DELETE läsas med ISAM READ. Om medskickad "sträng" ej överensstämmer med posten från föregående läsning fås ERROR 123. Detta används för att kontrollera eliminering av poster i ett Multi User-system

FELMEDDELANDEN I ISAM

Åtta nya felmedelanden har tillkommit i och med ISAM-paketet.

120 - Nyckel finns ej
121 - Dublettnyckel
122 - Felaktig nyckel
123 - Fel vid kontrolläsning
124 - Index finns ej
125 - Felaktig postlängd
126 - Fel ISAM-fil version
127 - Ej använd felkod
128 - Slut på minne i centralen
129 - Reserverad felkod

PROGRAMEXEMPEL MED ISAM

Vi bestämmer oss för tex följande poststruktur.

Fält	Benämning	Längd	Typ
1	Artilkelnr	15	ASCII
2	Benämning	20	ASCII
3	Saldo	2	INTEGER
4	In-pris	8	DOUBLE
5	Ut-pris	8	DOUBLE
6	Försäljning	8	DOUBLE

Postlängd = 61

Artikelnr och benämning skall bli våra söknycklar.

För att skapa ISAM-filen kör vi programmet CREINDEX på ISAM-disketten.

CREINDEX ställer flera frågor och vi besvarar frågorna med data för ovanstående lagerregister.

```
** Skapa ISAM-filer Ver x.xx **

 * Skapa filer *

Namn på nyckelfil ? ARTIKLAR
Namn på huvudfil  ? ARTIKLAR
Postlängd         ? 61

 * Skapa index nr 1 *


Namn på index          ? ARTNR
Startposition          ? 1
Längd på index         ? 15
Indextyp (B,A,I,F,D)   ? A
Dublett-nycklar (J/N)  ? N

 * Skapa index nr 2 *

Namn på index          ? BENÄMN
Starposition           ? 16
Längd på index         ? 20
Indextyp (B,A,I,F,D)   ? A
Dublett-nycklar (J/N)  ? J

 * Skapa index nr 3 *

Namn på index          ? <RETURN>
```

Filerna är nu klara för användning. I bilaga 2 ges små enkla exempel på hur ISAM-instruktionerna kan användas för att bearbeta lagerfilen. Exemplen är EJ kompletta ifråga om felhantering.

3. DATALAGRING I PRIMÄRMINNET

3.1 Aritmetik

3.1.1 Variabeltyper och precision

Variabeltypen anger vad för typ av värden som kan tilldelas en variabel. Precisionen anger med hur många siffror datorn arbetar.

De variabeltyper som finns i BASIC är:

* HELTAL	dessa kan vara:	* Enkla variabler * Vektorer * N-dimensionell matris
* FLYTTAL	dessa kan vara:	* Enkla variabler * Vektorer * N-dimensionell matris
* STRÄNGAR	dessa kan vara:	* Enkla variabler * Vektorer * N-dimensionell matris

Vid flyttal kan man också ange med hur många siffror datorn skall arbeta (instruktionerna SINGLE och DOUBLE). Man bör alltid använda SINGLE utom då en utvidgad precision är av största vikt. SINGLE leder nämligen till ett avsevärt snabbare och mindre program.

Vid SINGLE arbetar datorn med 7 siffror och vid DOUBLE med 16.

Man bör alltid använda heltal där det går. Heltalen medger snabbare exekveringstid och tar dessutom upp mindre plats i minnet jämfört med flyttalen.

Det är inte bara i symboltabellen som minnesbesparningarna sker utan även i internkoden. Observera att det är skillnad mellan olika tal vad gäller hur mycket plats talet tar upp. Sålunda tar talen:

```
    0 - 16        upp 1 (EN!) byte
   17 - 65535     upp 3 byte
 (-0) - (-16)     upp 2 byte
(-17) - (-65535)  upp 4 byte
```

Satsen
```
10 IF PEEK(-747) THEN ;´ERROR´
```
tar upp mer plats än
```
10 IF PEEK(64789) THEN ;´ERROR´
```
som ger exakt samma resultat.

Även vissa flyttals-konstanter lagras ekonomiskt. Binära tal lagras snålare än de ej binära.

Med andra ord; alla tal som kan skrivas som 2 ** n där n är ett heltal (-128) - (+127) lagras i två byte.

Exempel:
Satsen 10 A.=4. platsbehov är:

4 byte för radnumret
3 byte för A.=
2 byte för 4.

Summa 9 byte

Satsen 10 A.=0.1 platsbehov är:

4 byte för radnumret
3 byte för A.=
9 byte för 0.1

Summa 16 byte

Tal som man lätt kan beskriva i binär form, tex 10 (1010 binärt), tar upp relativt liten plats (3 byte), medan 0.1 tar upp 9 byte.

Exempel:
A.=1./10. tar mindre plats än A.=0.1
och
B.=1./3. tar mindre plats än B.=0.333...

Dock blir exekveringstiden längre om man använder det förra skrivsättet.

* HELTAL

Värdet av ett heltal lagras i två byte i tvåkomplementform. Med två byte kan ett binärtal skrivas inom talområdet 0 till 65535. I tvåkomplementform blir talområdet i stället -32768 till +32767 då tvåkomplementformen använder den mest signifikanta biten, dvs bit 15, till att indikera om talet är negativt eller positivt. Är talet negativt är biten ettställd. Detta gör att man får vara försiktig i sina program så att variabeln ej "slår runt", dvs går från ett negativt värde till ett positivt och vice versa. Detta beror på att det skall vara möjligt att räkna med adresser i talområdet 0 till 65535.

I symboltabellen tar en enkel heltalsvariabel upp 6 byte.

* FLYTTAL

Ett flyttal är ett tal på formen:

+/- Mantissa * 10 ** Exponent

I datorn lagras ett enkelt flyttal i 4 byte vid SINGLE och i 8 byte vid DOUBLE. Till skillnad från ABC80, som använde BCD-aritmetik, använder ABC800 binäraritmetik. I datorn ser därmed ett flyttal ut på följande sätt:

byte 1 Exponenten
byte 2-4/8 Mantissan

Beräkning av korrekt exponentvärde ur exponentbyten sker med hjälp av formeln:

Exponentvärde = Exponentbyten - 128

Byte 2 skiljer sig från de övriga i mantissan genom att den mest signifikanta biten anger om mantissan är negativ eller ej. Är biten ettställd är mantissan negativ annars positiv.

För att spara plats så lagras inte den högsta biten i mantissan då den alltid pga av normalisering är "ett" (den biten används i stället för att visa tecknet på mantissan enligt ovan). Ett undantag från detta är då flyttalet är noll (0) vilket indikeras genom att exponentbyten är noll.

I symboltabellen upptar ett flyttal 8 byte vid SINGLE och 12 byte vid DOUBLE.

* STRÄNGAR

Strängar används för att kunna hantera valfri text. Strängar används också vid numeriska beräkningar då beräkningar med ett stort antal siffror är möjligt på strängar mha ASCII-aritmetiken (se kap 3.1.2).

3.1.2 ASCII-aritmetik

ASCII-aritmetiken är mycket användbar vid beräkning av stora tal och där stor noggranhet behövs. För grundläggande ASCII-aritmetik hänvisas till "ABC om BASIC" och "AVANCERAD PROGRAMMERING PÅ ABC 80".

På ABC800 har det tillkommit en del nyheter i ASCII-aritmetiken vilka redovisas nedan.

* Antalet siffror har utökats till 125.

* Exponent tillåts i indata.

En annan nyhet är att om beräkning begärs med negativt antal decimaler tolkas detta som antal gällande siffror.

Detta medför:

```
;ADD¤(´10.55´,´10.55´,2)      2 decimaler
21.10

;ADD¤(´10.55´,´10.55´,-2)     2 gällande
21                            siffror

;ADD¤(´10.55´,´10.55´,-1)     1 gällande
20                            siffra
```

Några fel finns dock:

* Om addition eller subtraktion utförs med 0 som första term och andra termen är mindre än 1.E-Q, där Q är större än antalet gällande siffror + 2, faller andra termen bort.

  Exempel:

```
  ;ADD¤(´0´,´.1E-4´,-2)     2 gällande
  0                         siffror!!
```

* Utskrift med PRINT USING och exponentformat (ÜÜÜÜ) och exponent över 99 ger helt fel utskrift i första BASIC-versionen. I nya BASICen blir exponenten utskriven riktigt men exponentdelen tar upp 5 tecken istället för 4.

* Exponenter >126 eller <-127 lagras felaktigt.

3.2 Stränghantering

I detta avsnitt behandlas nyheterna i stränghanteringen som finns på ABC800 gentemot ABC80. För grundläggande stränghantering refereras till "ABC om BASIC" och "BASIC II boken".

NYHETER

Stränghanteringen på ABC800 har några mycket användbara nyheter.

* MID¤(, ,) kan stå i vänsterledet i ett stränguttryck.

  Exempel:

  MID¤(A¤,5,2)="AB"

  Detta förenklar tilldelning av en delsträng.
  Jmfr A¤=LEFT¤()+"AB"+RIGHT¤().

  OBS! Strängens längd får ej ändras (A¤ i exemplet ovan).

* VARPTR() Funktion som returnerar adressen till en variabels dataarea. Fungerar på alla datatyper men är mest användbar på strängvariabler.

* VAROOT() Funktion som returnerar adressen till en variabels parameterblock. Denna funktion är också mest användbar på strängvariabler.

VARPTR, VAROOT

VAROOT ger adressen till en variabels parameterblock. I strängvariabelns fall ser parameterblocket ut på följande sätt (se även kap 5.3.5):

VAROOT pekar hit ->	Dimensionerad längd	(l)	Adress
	Dimensionerad längd	(h)	Adress+1
	Adress till dataarean	(l)	Adress+2
	Adress till dataarean	(h)	.
	Aktuell längd	(l)	.
	Aktuell längd	(h)	.

Av ovanstående figur framgår att en strängvariabels dimensionerade längd kan fås mha:

```
Dimlen=PEEK2(VAROOT(Sträng¤))
```

En pekare till dataarean för strängvariabeln fås mha:

```
Varpntr=PEEK2(VAROOT(Sträng¤)+2)
```

Detta motsvarar:

```
Varpntr=VARPTR(Sträng¤)
```

Aktuell längd kan fås med:

```
Aktlen=PEEK2(VAROOT(Sträng¤)+4)
```

Detta motsvarar

```
Aktlen=LEN(Sträng¤)
```

Att komma åt denna viktiga information på detta enkla sätt är mycket användbart. Några exempel:

Exempel 1:
Från en Random Access-fil läses poster med en längd av 50 tecken. En post består av följande 3 delar:

```
Namn     20 tecken pos 1-20
Adress   20 tecken pos 21-40
Telefon  10 tecken pos 41-50
```

Ett enkelt sätt att dela upp posten är följande:

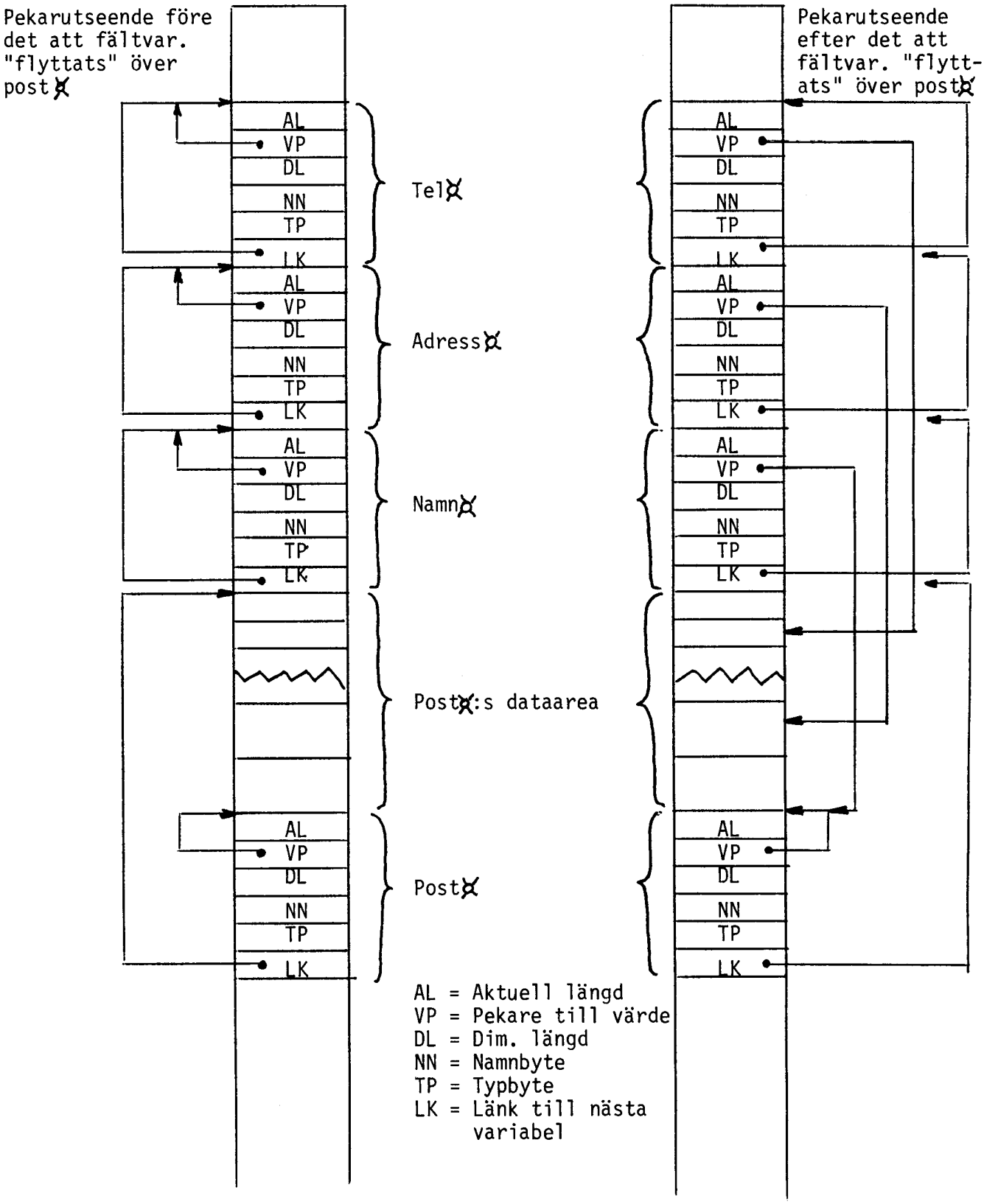
Dimensionera de olika variablerna.

```
10 DIM Post¤=50,Namn¤=0,Adress¤=0,Tel¤=0
```

De tre fältvariablerna dimensioneras till länden 0 för att inte ta upp onödig plats. "Flytta" därefter de tre fältvariablernas imaginära dataareor, som har längden 0, till sin respektive plats i dataarean för postvariabeln Post¤. Ändra samtidigt dimensionerad och verklig längd till längden på det fält som variabeln skall innehålla.

```
10 DIM Post¤=50,Namn¤=0,Adress¤=0,Tel¤=0
20 Post¤=SPACE¤(50)
30 Adr=VARPTR(Post¤)
40 POKE VAROOT(Namn¤),20,0,Adr,SWAP%(Adr),20,0
50 POKE VAROOT(Adress¤),20,0,Adr+20,SWAP%(Adr+20),20,0
60 POKE VAROOT(Tel¤),10,0,Adr+40,SWAP%(Adr+40),10,0
```

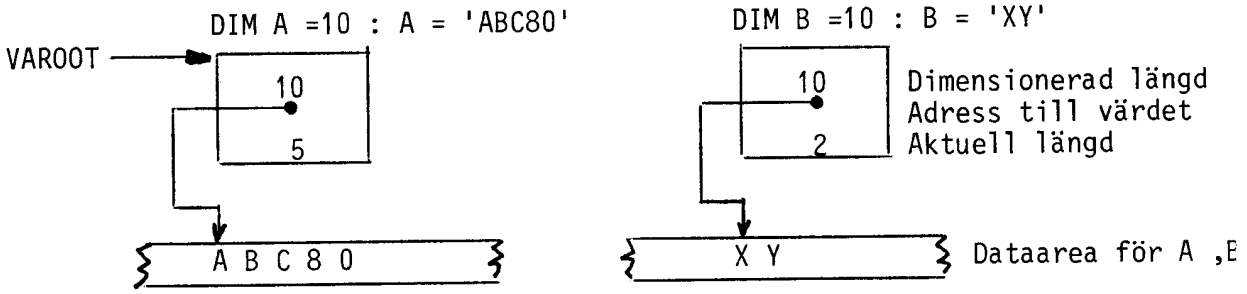
På rad 30 tilldelas Adr adressen till Post¤:s dataarea. Sedan förändras parameterblocken för de tre fältvariablerna så att de "flyttas" till Post¤. "Poka" dit dimensionerad längd, adress till dataarean och aktuell längd. Observera att alla värden ligger "swappade" (högsta byten sist). Sätt aktuell längd till det maxvärde som respektive fält har.


Pekarutseende före det att fältvar. "flyttats" över post¤
Pekarutseende efter det att fältvar. "flyttats" över post¤
AL
VP
DL
NN
TP
LK
Tel¤
Adress¤
Namn¤
Post¤:s dataarea
Post¤
AL = Aktuell längd
VP = Pekare till värde
DL = Dim. längd
NN = Namnbyte
TP = Typbyte
LK = Länk till nästa variabel

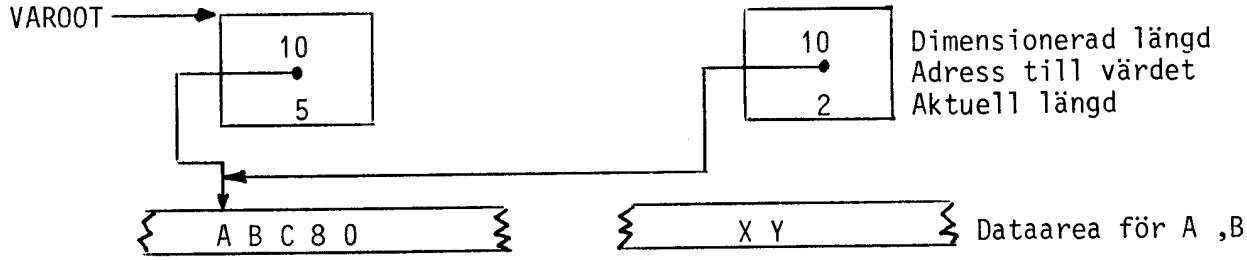
Vid läsning av en post till Post¤ hamnar automatiskt namn,adress och telefon i respektive variabler utan att programmet behöver dela upp Post¤ med hjälp av MID¤(). Likaså får Post¤ automatiskt rätt värde när någon av fältvariablerna tilldelas nytt innehåll. Vid utskrift av Post¤ behöver ej Post¤ skapas genom konkatenering av fältvariablerna. Observera att vid tilldelning av fältvariablerna måste dess maximala längd alltid fyllas upp, tex med blanktecken.

Vid kommunikation med rutiner i assembler kan adressen till en eller flera BASIC-variabler lätt fås fram med VARPTR. Se vidare kap 6.

En assemblerrutin kan läggas in i en sträng med tex
Code¤=CHR¤(x,y,...,201)
och sedan anropas med CALL(VARPTR(Code¤)). Se kap 6.



A och B är två 10-teckens strängar med olika innehåll. Genom att förändra B :s parameterblock kan B och A ha en gemensam dataarea.



A och B har nu en gemensam dataarea. Observera att om vi förändrar A :s aktuella längd kommer inte B att påverkas.

Vid applikationer där gemensamma dataareor kan vara intressanta, bör B dimensioneras till längden 0 (noll). Detta beror på att B :s gamla dataarea ej kan nås, utan tar endast upp onödig plats.

DIM

Vid behov av strängmatriser med olika dimensionerad längd på varje enskilt element kan följande metod tillämpas:

Dimensionera strängmatrisen UTAN att ge någon dimensionerad längd.

```
DIM A¤(20,20)
```

Tilldela de element som skall ha en längd över 80 tecken.

```
A¤(x,y)=SPACE¤(Längd) : A¤(xn,yn)=SPACE¤(Ln)
```

Redimensionera matrisen till den längd som övriga element i matrisen skall ha.

```
DIM A¤(20,20)=5
```

Alla icke tidigare dimensionerade element i matrisen får nu längden 5.

LÅNGA STRÄNGAR

När en enskild sträng blir så stor att tilldelning eller konkatenering (sammanslagning) inte är möjlig utan att minnet blir fullt, kan VARPTR och VAROOT utnyttjas för vidare bearbetning.

Vi börjar med att dimensionera en sträng till dess maximala längd.

```
DIM S¤=xxxx
```

Nu kan rutiner konstrueras för att lägga in respektive ta bort delar av strängen utan att påverka storlekskontroller i BASICen.

Vill vi tilldela S¤ en eller flera andra strängars värden gör vi det med följande metod:

* Hämta adressen till S¤:s dataarea.

* Hämta adressen till den tilldelande strängens dataarea.

* Flytta dataarean med assemblerinstruktionen LDIR.

* Justera aktuell längd i S¤ med hjälp av VAROOT-funktionen.

På samma sätt kan vi eliminera delar av strängen eller fylla delar av strängen med konstantvärden, se listningarna på funktionerna FNFILL, FNERASE och FNINSERT i bilaga 2.

4.1 Ljudgeneratorn

På ABC800 består ljudgeneratorn av ett D-register kopplat som en bistabil vippa. Vippan styrs via inport nr 5.

Denna enkla ljudgenerator, till skillnad från ABC80:s programmerbara, medför att vi måste skapa programrutiner för att åstadkomma ljud.

ENKLA LJUD

I de fall vi bara behöver ett "pip" styr vi ljudgeneratorn med följande kod.

BASIC:
```
PRINT CHR¤(7);
```

ASSEMBLER:
```
TUT   LD    HL,PIP
      LD    BC,1
      CALL  000BH
      RET
PIP   DEFB  07
```

SAMMANSATTA LJUD

När vi behöver variera längd och frekvens behöver vi lite annorlunda rutiner. Frekvensen bestämmer vi genom att variera tiden mellan varje gång vippan byter läge och längden styr vi tex med antalet svängningar som vippan skall göra.

BASIC:
```
! Integer mode
FOR L=1 to Längd
  Z=INP(5)
  FOR F=1 to Tid
  NEXT F
NEXT L
```

ASSEMBLER:
```
TUT   EQU   *
      LD    B,D
      LD    D,00H
;
LOOP  EQU   *
      IN    A,05H
      PUSH  BC
;
WLOOP EQU   *
      DJNZ  WLOOP
      POP   BC
      DEC   DE
      LD    A,D
      OR    E
      JR    NZ,LOOP
      RET
```

Inparameter till assemblerrutinen är reg D och E. D=frekvens och E=längd.

Assemblerrutinen kan lätt läggas in i ett BASIC-program på följande sätt:

```
10 DEF FNTut(Frek,Längd) LOCAL Tut¤=15
20   Tut¤=CHR¤(66,22,0,219,5,197,16,254,193,27,122,179,32,
         245,201)
30   RETURN CALL(VARPTR(Tut¤),Frek*256+Längd)
40 FNEND
```

Lite data för "orgelspel". Tonerna är bara inbördes stämda.

Ton	Längd	Frek
b	200	242
H	205	228
C	210	214
C#	215	204
D	220	190
D#	225	180
E	230	170
F	235	160
F#	240	150
G	245	143
G#	250	136
A	255	128
B	255	120
H	255	114
C	255	107
C#	255	102
D	255	95
D#	255	88
E	255	85
F#	255	80
G	255	73

Vi försöker göra en lång ton, tex

```
FOR I=1 TO 1000
  Z=INP(5)
NEXT I
```

Vi kommer att höra ett "plick" varje sekund i tonen. Detta "plick" kommer från klockrutinen som varje sekund behöver lite längre tid på sig för att uppdatera klockan. Detta kan vi komma ifrån genom att lägga in ett DI (Disable Interrupt) före och ett EI (Enable Interrupt) efter rutinen som gör tonen.

OBSERVERA att om tonen är längre än 10 ms kan klockan komma att gå fel och om tonen är längre än 20 ms så kommer klockan att gå fel. Anledningen är att klockan uppdateras varje 10:e ms. Om interruptet är avstängt missas en uppdatering.

4.2 VDU (80 tecken)

ABC800 levereras med olika bildskärmalternativ. Detta avsnitt behandlar den monochroma 80-teckensvarianten.

När ABC800 är utrustad med en 80-teckens skärm (eg 80-teckens VDU-kort i datorenheten) sköts representationen på skärmen av en speciell VDU-krets (MC 6845). Se LUXOR servicemanual.

VDU-kretsen genererar synk-signaler till bildskärmen samt styrsignaler för minnesavkodning och bildgenerering. Kretsen innehåller olika register för generering, tex antal rader som skall visas, från vilken adress i minnet visning skall ske etc.

Kretsen är även förberedd för användning av ljuspenna.

Kretsen kan programmeras direkt från BASIC med

```
OUT 56,reg,57,data ! Skrivning
```

Där "reg" är det register vi vill skriva till.
"data" är det värde vi vill skriva in i registret.

Detta kan utnyttjas till ett antal olika saker:

1. Modifiera utseendet på cursor.

2. Flytta "bilden" horisontellt/vertikalt på bildskärmen.

3. Öka radantalet till 25.

Modifiera cursor

Register 10 och 11 innehåller information för cursor. Register 10 används normalt av BASIC:en medan register 11 kan ändras för att man tex ska få "fet" cursor.

Exempel:

```
OUT 56,11,57,9  !  Ger "fet" cursor (2 linjer)
```

Flytta "bilden"

Med olika värden i register 2 (mellan ca 80 och 100) flyttar man bilden horisontellt.

Och med olika värden i register 5 (0 - 31) sker flyttning vertikalt.

OBS!
I vissa fall kan "synk"-problem uppstå.

Se vidare programmet CRTADJ i bilaga 2, som innehåller ett komplett programexempel för att "flytta" bilden i alla riktningar.

Öka radantalet till 25

ABC800s bildminne består av 2kbyte (2048 byte). En enkel beräkning ger att 2048/80 = 25.6, dvs drygt 25 rader. Det outnyttjade utrymmet kan normalt sett ej utnyttjas men med en speciell programsekvens kan VDU-kretsen initieras för att även skriva ut denna del. I programsekvensen lägger vi också ut den önskade textinformationen med tex ;CHR¤(12), som ej kan raderas.

Exempel:

```
10000 DEF FNInirow25(Text¤) LOCAL Vdu¤=0
10010   OUT 56,6,57,25
10020   POKE VAROOT(Vdu¤),80,0,32688,127
10030   Vdu¤=Text¤
10040   RETURN -1
10050 FNEND
10060 !
10070 Z=FNInirow25(´DETTA ÄR RAD 25 ELLER RAD -1´)
```

VDU-kretsens register

Reg nr	Reg namn	Normalvärde	
0	Horisontal total	127	
1	Horisontal display	80	(tecken per rad)
2	Horisontal sync delay	100	
3	Horisontal sync width	9	
4	Vertical total	30	
5	Vertical adjust	4	
6	Vertical display	24	(antal rader)
7	Vertical sync pos	28	
8	Interlace mode	0	
9	Max line adress	0	
10	Cursor start in line	40	
11	Cursor end in line	8	
12	Start adress VDU mem (h)	120	(adress bildminne)
13	Start adress VDU mem (l)	0	(-"- -"-)
14	Cursor adress (h)	120	
15	Cursor adress (l)	0	
16	Light pen reg (h) read only	?	
17	Light pen reg (l) read only	?	

4.3 Högupplösningsgrafiken

4.3.1 Allmänt

Högupplösningsgrafiken (härefter förkortad HR-grafiken) är en option till ABC 800 och kan användas på både M och C modellen.

HR-grafiken kan, om så önskas, användas samtidigt med normalt bildminne. Bilden består av 240*240 "pixels" där varje pixel är direkt adresserbar. Varje pixel kan anta fyra olika (logiska) färger (0-3). Vilka ("fysiska") färger 0-3 skall representera bestäms med ett färgvalskommando.

HR-bilden ligger lagrad i ett 16 Kbyte (16384 byte) minne med startadress 0. Detta medför att åtkomst av HR-minnet är besvärlig pga att BASIC-tolken upptar samma minnesutrymme. PEEK eller POKE kan inte användas.

De sätt som finns att adressera HR-minnet är antingen genom att använda de nya BASIC-instruktionerna eller att använda assembler, vilket kommer att visas längre fram.

4.3.2 BASIC-instruktioner

FGCTL färggrupp	Väljer angiven färggrupp. Färgrupp kan anta värden mellan 0-127 och 128-255. Det senare området medför att den vanliga textbilden väljs bort. Vilka färger varje färggrupp representerar finns beskrivet i bilaga 8 och BASIC II manualen.
FGFILL x,y<,färgnr>	Fyller en rektangel från föregående position till position x,y med färg färgnr. Om färgnr utelämnas antas senast använda färg.
FGLINE x,y<,färgnr>	Drar en rät linje från föregående position till position x,y. Även här kan färgnr utelämnas.
FGPAINT x,y<,färgnr>	Fyller en sluten yta med färg färgnr. Den yta som skall målas innesluts lämpligen med FGLINE. Färgnummer kan utelämnas även här.
FGPOINT x,y<,färgnr>	Sätter en "pixel" i position x,y med färg färgnr. Detta är den instruktion man lämpligen startar med. Färgnummer kan utelämnas.
FGPOINT(x,y)	Returnerar färgen på pixel i position x,y.

Exempel:

```
10 FGPOINT 0,0,0          ! Sätt aktuell position till 0,0
                          med färg 0
20 FGFILL 239,239         ! Fyll till position 239,239 med
                          färg 0
                          ! Obs rad 10-20 är enklaste sättet
                          ! att blanka HR-bilden (Jmfr
                          ;CHR¤(12)).
30 FGCTL 4                ! välj färggrupp 4
40 FGPOINT 50,50,2        !
50 FGLINE 50,150          !
60 FGLINE 150,150         !
70 FGLINE 150,50          !
80 FGLINE 50,50           !
90 FGPAINT 75,75,2        ! Fyll rektangel med färg 2
```

LITE KURIOSA

HR-bildens startposition på skärmen kan bestämmas med kommandot OUT 6,radnr där radnummer kan variera mellan 0-255.

OBS!
Port 6 används av vissa på marknaden förekommande minneskort och kan då ej användas tillsammans med HR-minnet. En felaktig användning kan orsaka att programmet "dyker".

4.3.3 Animation

Rörliga bilder går också att åstadkomma med hjälp av lite knep och knåp. Färggrupperna mellan 72-127 och 200-255 är avsedda för detta. Färggrupperna används två och två, ex.vis 72,73 där tex 72 har färg 2=röd och 3=svart och grupp 73 har färg 2=svart och 3=röd.

Sätt FGCTL 72 och rita med färg 2 utan att det syns något på skärmen. Växla sedan till FGCTL 73 och visa den nyss ritade bilden. Rita ånyo med färg 1, som inte syns, och växla till FGCTL 72 för att visa den sist ritade bilden osv osv.

Ett problem är att radera de "gamla" bilderna (genom att rita med färg 0) utan att förstöra den bild som visas på skärmen. Detta är löst i BASIC:en genom att man sätter ´skyddsbitar´ i färgnummret. Varje pixel upptar två bitar i HR-minnet. Med hjälp av två bitar kan vi beskriva talen 0-3.

RITA MED FÄRG	SKYDDA FÄRG	ANVÄND FÄRG NR
1	2	256*2+1 = 513
2	1	256*1+2 = 258
0	2	256*2+0 = 512
0	1	256*1+0 = 256

PRINCIP FÖR ANIMERING

1 Blanka HR-bilden.
2 Välj en lämplig färggrupp.
3 Rita en bild i samma färg som färggruppens bakgrundsfärg.
4 Byt färggrupp så att bilden framträder.
5 Rita nästa bild i den nya färggruppens bakrundsfärg.
6 Byt färggrupp så att den nya bilden framträder.
7 Radera den gamla bilden (som ej framträder) genom att rita med färg 0 och rita en ny bild i bakgrundsfärgen.
8 Upprepa från 6.

OBS!
Rita alltid med skyddsbitarna satta enligt ovanstående tabell.

Ett litet programexempel på animering finns listat i bilaga 2.

4.3.4 Exempel

HR-minnet upptar 16 kbyte (16384 byte) med startadress 0. Varje pixel tar upp två bitar. HR-minnet har följande indelning:

Adress	Innehåll
0-59	Pixelrad 0,239-239,239 (x,y - x,y)
60-63	Ej använda
64-123	Pixelrad 0,238-239,238 (x,y - x,y)
124-127	Ej använda
.	
.	
.	
15296-15355	Pixelrad 0,0-239,0 (x,y - x,y)
15356-16383	Ej använda

I och med att HR-minnet tar upp samma plats i adressrummet som BASIC-interpretatorn, går det inte att göra POKE/PEEK (som i bildminnet). Enda sättet att skriva/läsa information i HR-minnet (förutom med de reserverade BASIC-instruktionerna) är att använda assembler.

När man exekverar kod i de adresser bildminnet upptar (där ligger optionsprommet som innehåller rutinerna för FG...) väljs HR-minnet automatiskt in.

På adress 32765 ligger assemblerinstruktionerna LDIR följt av RET. Med hjälp av detta kan vi bygga upp följande rutiner för läsning resp. skrivning i HR-minnet:

```
LÄS   EQU  *              SKRIV  EQU  *
      LD   HL,HR.ADR             LD   HL,VAR.ADR
      LD   DE,VAR.ADR            LD   DE,HR.ADR
      LD   BC,ANTAL              LD   BC,ANTAL
      JP   32765                 JP   32765
```

I BASIC:

```
100 Läs¤=CHR¤(33)+CVT%¤(Hradr)+CHR¤(1)+CVT%¤(Antal)+
    CHR¤(195,253,127)
110 Dummy=CALL(VARPTR(Läs¤),VARPTR(Hr¤)

100 Skriv¤=CHR¤(33)+CVT%¤(VARPTR(Hr¤))+CHR¤(1)+CVT%¤(antal)+
    CHR¤(195,253,127)
110 Dummy=CALL(VARPTR(Skriv¤),Hradr)
```

Dessa grundrutiner använder vi oss av när vi vill spara hela HR-bilder på fil, antingen diskett/kassett eller skrivare (som klarar HR-grafik). Se funktionerna FNHrput, FNHrget FNHrsave, FNHrload och FNHrerase i bilaga 2.

5. BASIC-TOLKEN

5.1 Systemvariabler

5.1.1 Internvariabler

Här följer en förteckning över de adresser som används internt av BASIC-interpretatorn samt det programinformationsblock som föregår varje program.

Ofta används adresserna direkt ur nedanstående lista. Detta kan dock medföra problem vid framtida användning. Det korrekta sättet är att referera till indexregister IY eller SYS(10), som alltid pekar til aktuellt BASIC-blocks början (0FF00H) och läsa minnet med offset (00H-80H) från detta register.

Exempel:

Assemblerrutinens utseende (Finns i Asm¤)

```
PUSH  IY
POP   HL
ADD   HL,DE
RET
```

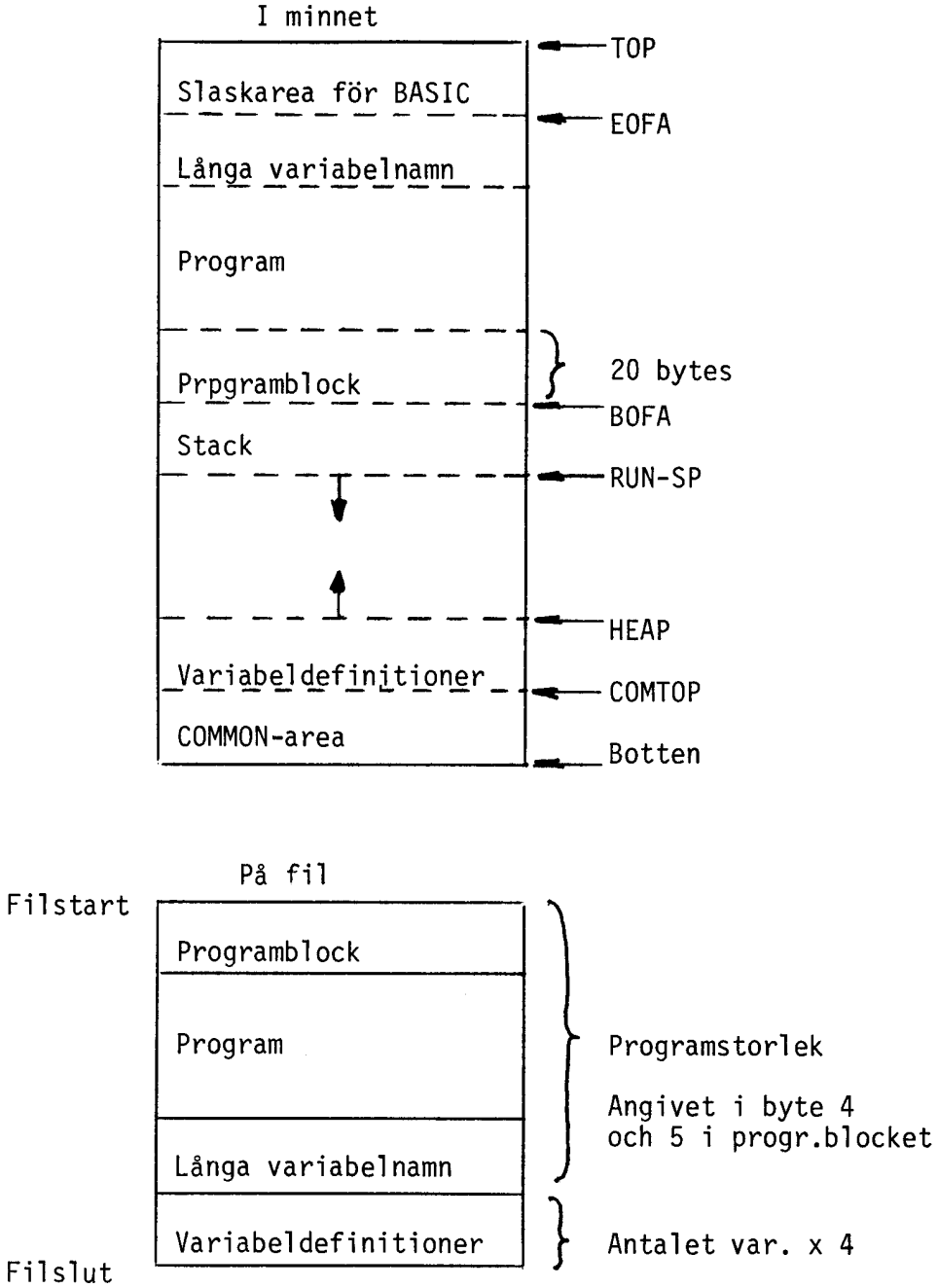
```
10000 DEF FNIy(Offset)
10010   RETURN SYS(10)+Offset
10020 FNEND
```

Detta medför att alla systemvariabler för BASIC är refererbara (dessa har prefix Y i listan). För att nå värden utanför denna lista måste diverse länkar i listan användas.

Exempel:

```
Radbredd=PEEK(PEEK2(FNIy(64))+8)    eller  Radbredd=PEEK(65364)
Kolumn=PEEK(PEEK2(FNIy(64))+6)      eller  Kolumn=PEEK(65362)
Rad=PEEK(PEEK2(FNIy(64)+7)          eller  Rad=PEEK(65363)

Bofa=FNIy(6)
Eofa=FNIy(8)
```



BASIC-programmets utseende i internminnet och på fil

Adr D	Adr H		Label	Init	Kommentar
65280	FF00				
65281	FF01				
65282	FF02				
65283	FF03				
65284	FF04	W	Y.ERLOB		Lokal bas vid ON ERROR
65285	FF05				
65286	FF06	W	Y.BOFA		Pekare till program-
65287	FF07				informationsblocket
65288	FF08	W	Y.EOFA		Pekare till sista byten
65289	FF09				i BASIC-programmet
65290	FF0A	W	Y.HEAP		Pekare till första
65291	FF0B				lediga byte i minnet
65292	FF0C	W	Y.BOTM	00	Botten på BASIC-minnet
65293	FF0D			80	(RAM)
65294	FF0E	W	Y.TOP		Toppen av BASIC-minnet
65295	FF0F				(RAM)
65296	FF10	W	Y.TOPPRG		Högsta adr för programmet
65297	FF11				
65298	FF12	W	Y.CONTSP		Stackpekare att använda
65299	FF13				vid CONTINUE
65300	FF14	W	Y.RESSP		Stackpekare att använda
65301	FF15				vid RESUME
65302	FF16	W	Y.CMDSP		Stackpekare att använda
65303	FF17				i direktmode
65304	FF18	W	Y.LOCBAS		Lokal variabelpekare
65305	FF19				
65306	FF1A	W	Y.GENEND		
65307	FF1B				
65308	FF1C				
65309	FF1D	B	Y.FL		Diverse flaggor *2
65310	FF1E	B	Y.PREC		Flyttalsprecision (4/8)
65311	FF1F	B	Y.DIGITS		DIGITS-värde (1-)
65312	FF20	B	Y.ASCSAV		
65313	FF21	B	Y.ASCPRE		
65314	FF22	B	Y.DEFLOW		OPTION BASE-värde
65315	FF23	B	Y.INT		Interrupt byte *5
65316	FF24	W	Y.STACK		
65317	FF25				
65318	FF26	B	Y.PRSTAT		Programstatus *3
65319	FF27	B	Y.XQS		Exekveringsstatus *4
65320	FF28	W	Y.CURDEF		
65321	FF29				
65322	FF2A	W	Y.FORCH		
65323	FF2B				
65324	FF2C	W	Y.VARTB		
65325	FF2D				
65326	FF2E	W	Y.VARBAS		Start på variabellista
65327	FF2F				
65328	FF30	W	Y.COMTOP		Toppen på COMMON-
65329	FF31				variablerna
65330	FF32	W	Y.COMCS		Checksumma för COMMON-
65331	FF33				variablerna
65332	FF34	B	Y.USERCS		Senaste kanalval
65333	FF35	B	Y.TRCLU		Logisk enhet vid TRACE
65334	FF36	W	Y.TRCVAR		Sist ändrade variabeln
65335	FF37				
65336	FF38	W	Y.VAROOT		Sist passerade VAROOT

Adr D	Adr H		Label	Init	Kommentar
65337	FF39				
65338	FF3A	W	Y.IPSAV		Instruktionspekare
65339	FF3B				
65340	FF3C	W	Y.RDPTR		Pekare till aktuell position i
65341	FF3D				DATA-sats
65342	FF3E	W	Y.RDPTR1		Pekare till aktuell DATA-sats
65343	FF3F				
65344	FF40	W	Y.LUCH		Pekare till första öppna fil
65345	FF41				(CON:)
65346	FF42	W	Y.ONERR		Pekare till ON ERROR-rutin
65347	FF43				
65348	FF44	B	Y.ERRCOD		ERRCODE
65349	FF45	W	Y.SPSAV		Temporär lagring av reg SP
65350	FF46				
65351	FF47	F	Y.RND		Gammalt slumptal
65352	FF48				
65353	FF49				
65354	FF4A				
65355	FF4B				
65356	FF4C	W		00	Pekare till nästa fil-
65357	FF4D			00	parameterlista (IX)
65358	FF4E	B	LU.LU	00	Filnummer
65359	FF4F	B	LU.STAT	07	Status
65360	FF50	W	LU.DCB		Entrypoint
65361	FF51				
65362	FF52	W	LU.POS		Kolumn (0-Radbredd)
65363	FF53				Rad (0-23)
65364	FF54	W	LU.WID	28/50	Radbredd
65365	FF55			00	
65366	FF56	B	LU.FC		Sist utförda operation
65367	FF57	W	LU.ISAMB		ISAM-block
65368	FF58				
65369	FF59	W	LU.EOF		Antal block i filen
65370	FF5A				
65371	FF5B	W	LU.BUFN		Nummer på aktuellt
65372	FF5C				block i bufferten
65373	FF5D	W	LU.RNDRC		Blocknummer för
65374	FF5E				random access
65375	FF5F	B	LU.RNDO		Random access buffert offset
65376	FF60	B	LU.BUFH		Buffertadress (H)
65377	FF61	B	LU.EXT		LU-block extension
65378	FF62	B	LU.LFT		DOSBUFnr

Adr D	Adr H		Label	Init	Kommentar
65379	FF63	S23	UTTLLU		LU-Block för temporära ope-
65380	FF64				tioner. Indelning lika med
65381	FF65				ovanstående.
65382	FF66				
65383	FF67				
65384	FF68				
65385	FF69				
65386	FF6A				
65387	FF6B				
65388	FF6C				
65389	FF6D				
65390	FF6E				
65391	FF6F				
65392	FF70				
65393	FF71				
65394	FF72				
65395	FF73				
65396	FF74				
65397	FF75				
65398	FF76				
65399	FF77				
65400	FF78				
65401	FF79				
65402	FF7A				
65403	FF7B	W	DEVTBA	5B	Pekare till länkad
65404	FF7C			11	enhetslista
65405	FF7D	W	STMTBA	06	Pekare till länkad
65406	FF7E			09	instruktionslista
65407	FF7F	W	FNKTBA	DC	Pekare till länkad
65408	FF80			06	funktionslista
65409	FF81	W	CTOBUF	00	Pekare till
65410	FF82			FD/FB	kassettbuffert 0
65411	FF83	W	XQTPTR	2E	Pekare till AUTOSTART-
65412	FF84			00	kommando
65413	FF85	W	CTRLCPTR	23	Pekare till CTRL-C flagga
65414	FF86			FF	
65415	FF87	S3	CMDUNSA	C3	Hopp till UNSAVE-rutin
65416	FF88			36	
65417	FF89			00	
65418	FF8A	S3	NMIENT	C3	NMI-interrupt entry
65419	FF8B			00	Hit sker hopp vid NMI
65420	FF8C			00	
65421	FF8D	S3	HRCLR1	C3	Släck högupplösningsgrafik
65422	FF8E			9C	(anropas av bla LIST-kommandot)
65423	FF8F			19	
65424	FF90	S3	CONSI	C3	Adress till Läs 1 tecken (GET)
65425	FF91			40	
65426	FF92			03	
65427	FF93	S3	KILLTYPE	C3	
65428	FF94			9C	
65429	FF95			19	
65430	FF96	S3	USERTRAC	C3	Användarrutin för
65431	FF97			BB	debugging
65432	FF98			2C	
65433	FF99	B	CASSPEED	28	Hastighet vid skrivning (CAS:)

Adr D	Adr H	Label	Init	Kommentar
65434	FF9A S3	DEBUGENT		
65435	FF9B			
65436	FF9C			
65437	FF9D S3	RST6ENT		RST 30H hoppar hit
65438	FF9E			
65439	FF9F			
65440	FFA0 S3	RST7ENT		RST 38H hoppar hit
65441	FFA1			
65442	FFA2			
65443	FFA3 S3	ISAMOPN		Ledigt för ISAM-OPEN
65444	FFA4			
65445	FFA5			
65446	FFA6 S3	O.MATARI		Ledigt för matris-
65447	FFA7			matematik (Evaluering)
65448	FFA8			
65449	FFA9 S3	MAKEMAT		Ledigt för matris-
65450	FFAA			hantering (kompilering)
65451	FFAB			
65452	FFAC W	ERRPTR		DOS-anrop för att hämta error-
65453	FFAD			text. A = Felnr
65454	FFAE B	CASBSY	00	
65455	FFAF			
65456	FFB0 S16	DARTVEK	87	DART-vektor
65457	FFB1		03	
65458	FFB2		87	
65459	FFB3		03	
65460	FFB4		6D	Interrupt-adress för tangent-
65461	FFB5		03	bordet
65462	FFB6		89	
65463	FFB7		03	
65464	FFB8		87	
65465	FFB9		03	
65466	FFBA		87	
65467	FFBB		03	
65468	FFBC		87	
65469	FFBD		03	
65470	FFBE		87	
65471	FFBF		03	
65472	FFC0 S16	SIOVEK	87	SIO-vektor
65473	FFC1		03	
65474	FFC2		87	
65475	FFC3		03	
65476	FFC4		87	
65477	FFC5		03	
65478	FFC6		87	
65479	FFC7		03	
65480	FFC8		87	
65481	FFC9		03	
65482	FFCA		87	
65483	FFCB		03	
65484	FFCC		87	
65485	FFCD		03	
65486	FFCE		87	
65487	FFCF		03	

Adr D	Adr H	Label	Init	Kommentar
65488	FFD0 S8	CTCVEK	87	CTC-vektor
65489	FFD1		03	
65490	FFD2		87	
65491	FFD3		03	
65492	FFD4		87	
65493	FFD5		03	
65494	FFD6		87	Interruptvektor för
65495	FFD7		03	klocka
65496	FFD8 S8	OPTRAM		Minne reserverat för
65497	FFD9			optionsrutiner
65498	FFDA			
65499	FFDB			
65500	FFDC W			Adress till OPTIONS-prommets
65501	FFDD			RAM-area (eg PR:/V24:)
65502	FFDE			
65503	FFDF			
65504	FFE0 W	TABPTR		DE-save
65505	FFE1			
65506	FFE2 B	KEYFLG		Tangentbordsflagga
65507	FFE3 B	KEYCHR		Tangentbordsbuffert
65508	FFE4 W	CASCHKAD		
65509	FFE5			
65510	FFE6 W	CASSUM		
65511	FFE7			
65512	FFE8 B	CASMODE		
65513	FFE9 W	CASADR		
65514	FFEA			
65515	FFEB W	CASACTBF		
65516	FFEC			
65517	FFED W	CRECNR		
65518	FFEE			
65519	FFEF B	SPT.YEAR		(0-99) Klocka
65520	FFF0 B	SPT.MNTH		(1-12)
65521	FFF1 B	SPT.DAY		(0-31) (0 = stopp klocka)
65522	FFF2 B	SPT.HOUR		(0-23)
65523	FFF3 B	SPT.MIN		(0-59)
65524	FFF4 B	SPT.SEC		(0-59)
65525	FFF5 B	SPT.TIC		(0-93)
65526	FFF6 B			Ufd-reset (nyare DOS)
65527	FFF7 W			Ufd-offset (Nyare DOS)
65528	FFF8			
65529	FFF9 B			Ufd-drive (Nyare DOS)
65530	FFFA			
65531	FFFB			
65532	FFFC			
65533	FFFD			
65534	FFFE			
65535	FFFF			

*2 Diverse flaggor

Bit	Label	Kommentar
7		
6		
5		
4		
3		
2	FL.SPSI	Blanktecken är signifikant
1	FL.XTND	EXTEND mode
0	FL.IMODE	Default INTEGER mode

*3 Program status

Bit	Label	Kommentar
7	PRS.NWER	Ny errorhantering används
6	PRS.HRG	HR-grafik används
5	PRS.HI	Program är högt i minne
4	PRS.FPCO	Flyttal finns i COMMON
3	PRS.FPVR	Flyttal är allokerade
2	PRS.DP	Dubbel precision
0	PRS.FIX	Fixed-up

*4 Exekveringsstatus

Bit	Label	Kommentar
7		
6		
5		
4		
3	XQS.TRC	Utskrift av radnr
2	XQS.RESUM	Inuti användares error-rutin
1	XOS.ONERR	ON ERROR-rutin finns
0	XQS.CONT	CONTINUE är tillåtet

*5 Interrupt-byte

Bit	Label	Kommentar
7		
6		
5		
4		
3	INT.STEP	Single step
2	INT.DIR	Direkt mode
1	INT.TRC	TRACE
0	INT.CTRC	CTRL-C flagga

* Programblock

Adr	Filadr		Namn	Förklaring
0	0	B	PR.BIN	Anger BASIC-version (143)
1	1	B	PR.SEGN	Segment nummer (Ej använd)
2	2	B	PR.PRSTAT	Program status (Se nedan)
3	3	B	PR.CSUM	Program checksumma(Ej använd)
4	4	W	PR.PRGSZ	Längd på programmet (inkl. detta block, exkl. variabel definitioner)
5	5			
6	6	W	PR.VARSZ	Skalära variabelareans storlek
7	7			
8	8	W	PR.VARAD	Adress till skalära variabel-arean (vid laddning)
9	9			
10	10	W	PR.ANTVR	Antal variabler
11	11			
12	12	W	PR.COMSZ	Storlek på COMMON-fältet
13	13			
14	14	W	PR.COMCS	Checksumma för COMMON-deklara-tioner
15	15			
16	16	W	PR.DEFCH	Adr till 1:a DEF-raden. Relativt SYS(11). 0 = Inga DEF finns
17	17			
18	18	W	PR.DATACH	Samma som ovan, men 1:a DATA-satsen
19	19			

Adressen i minnet fås med SYS(11) + adr.

5.1.2 DOS-variabler

Här följer en förteckning över de adresser som används internt av DOS.

Adr D	Adr H	Label	Kommentar
64768	FD00		Filens plats i biblioteket *1
64769	FD01		Disk-selectkod *2
64770	FD02 W		Aktuellt logiskt recordnr
64771	FD03		
64772	FD04 W		Logiskt recordnr i aktuell bitmapgrupp
64773	FD05		
64774	FD06 W		Aktuell bitmapgrupp
64775	FD07		
64776	FD08 W		Filens fysiska startrecord
64777	FD09		
64778	FD0A W		Högsta logiska recordnr i filen
64779	FD0B		
64780	FD0C W		Aktuell random access record i buffert
64781	FD0D		
64782	FD0E		Positionsräknare i bufferten
64783	FD0F		
64784	FD10		
64785	FD11		
64786	FD12 W		Adress till DOSBUF0 *3
64787	FD13		
64788	FD14		DOSBUFnr * 10H
64789	FD15		Disk-error kod *4
64790	FD16		
64791	FD17		
64792	FD18		"Retry"-räknare. Startvärde 5/3
64793	FD19		
64794	FD1A		
64795	FD1B		
64796	FD1C	A SAVE	OPEN/PREPARE/CLOSE/RESIZE
64797	FD1D W	BC SAVE	Spara värdet på BC-registret
64798	FD1E		
64799	FD1F W	DE SAVE	Sparar värdet på DE-registret
64800	FD20		
64801	FD21 W	ON ERR 35	ERROR 35. Adress från FD33 om 0
64802	FD22		
64803	FD23 W	ON ERR 36	ERROR 36. Adress från FD33 om 0
64804	FD24		
64805	FD25 W	ON ERR 37	ERROR 37. Adress från FD33 om 0
64806	FD26		
64807	FD27 W	ON ERR 38	ERROR 38. Adress från FD33 om 0
64808	FD28		
64809	FD29 W	ON ERR 39	ERROR 39. Adress från FD33 om 0
64810	FD2A		
64811	FD2B W	ON ERR 40	ERROR 40. Adress från FD33 om 0
64812	FD2C		
64813	FD2D W	ON ERR 41	ERROR 41. Adress från FD33 om 0
64814	FD2E		
64815	FD2F W	ON ERR 42	ERROR 42. Adress från FD33 om 0
64816	FD30		
64817	FD31 W	ON ERR 43	ERROR 43. Adress från FD33 om 0
64818	FD32		
64819	FD33 W	ON ERR DEF	Defaultadress vid error
64820	FD34		
64821	FD35		

64822	FD36
64823	FD37
64824	FD38
64825	FD39
64826	FD3A
64827	FD3B
64828	FD3C
64829	FD3D
64830	FD3E
64831	FD3F

*1 FILENS PLATS I BIBLIOTEKET (OFD00H)

Bit	7	6	5	4	3	2	1	0
	x	x	x	x	s	s	s	s

där x x x x anger bibliotekssektor (0-15)
s s s s anger biblioteksoffset (0-240 step 16)

*2 DISK-SELECT KOD (OFD01H)

Bit	7	6	5	4	3	2	1	0
	r	s	p	x	x	d	d	d

där r anger raderskyddad fil
s anger skrivskyddad fil
p anger om plats finns reserverad för filen
x x
d d d anger drivenummer

*3 DOSBUF0 ADRESS (OFD12H)

Anger startadress til DOS-buffertar. Fungerar som en läspekare i buffert0. Används bl.a. vid hantering av fil-mapparna.

*4 DISK-ERRORS (OFD15H)

Bit	Betydelse
7	Not ready (Luckan öppen)
6	Skivan är skrivskyddad
5	
4	Not found (Disk-fel)
3	CRC-fel (Checksumma-fel)
2	
1	Command error (Felaktig beordning till disk-controlern)
0	Busy

Olika bitkombinationer kan förekomma.

Intresserade hänvisas till Western Digitals, Technical MANUAL for FD179X för vidare information då disk-error byten återspeglar statusvärdet för diskcontroller-kretsen.

5.1.3 Olika DOS

På ABC800 förekommer en rikhaltig flora av olika DOS beroende på vilken diskettstation som används.

DOS	Klusterstorlek	Sektoradressering	Station
ABC 6-1X	1	x32	ABC830,DD82,DD84
800 8"	4	x1 (Special)	DD88
ABC 6-2X	4	x1	ABC832
ABC 6-3X	4	x1	ABC838
UFD 6.XX	32	x1	Winchester

Vilket DOS en dator är utrustad med kan man ta reda på på något av följande sätt. Det enklaste (och säkraste) sättet är att titta på beteckningen på DOS-prommet (pos K2). Men man kan också prova på följande sätt:

```
Är innehållet på adress
24678 =  195 -> ABC 6-2X  ABC 6-3X
24678 <> 195 -> ABC 6-1X  800 8"

Är 24678 = 195 är skillnaden:
24681 =  195 -> ABC 6-2X
24681 <> 195 -> ABC 6-3X
```

Skillnaden mellan ABC 6-1X och 800 8" (DD88) går ej att testa men kontroll om station är ansluten kan ske på följande sätt:

```
OUT 1,45
INP(1) =  255 -> Ingen ABC 6-1X station är ansluten.

OUT 1,44
INP(1) =  255 -> Ingen 800 8" station är ansluten.
```

I tabellen ovan märks en viktig skillnad. Sektornr skall ej tas x32 på nyare varianter av DOS.

En annan skillnad är att sektor 6 och 7 (bit-map) är flyttade till sektor 14 och 15 på DOS från ABC 6-2X.

På 800 8" (DD88) är sektoradressen kodad på följande sätt:

```
Bit  15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
      x  x  x  x  x  x x x x x x 0 0 0 x x

xxxxxxxxxx = sektornummer  Programmet skall "skifta"
                           bit 2 - 12 tre steg så
                           att ovanstående resultat
                           erhålles. Detta görs med
                           följande funktion:
```

```
DEF FNSectoradr(Sector,Cluster)=Sector/Cluster*32+(Sector AND
(Cluster-1))
```

NYA INTERNVARIABLER FÖR DOS

DOS ABC 6-2X och ABC 6-3X har ytterligare en uppsättning internvariabler förutom de ovan redovisade.

Följande enhetstabell läggs upp.

Enhet	Adress	Default (hex)	Namn
	+0	08/10	DRO offset
SF	+1	2C	Kanalnummer (Kortval)
	+2	04	Klusterstorlek
MF	+3	2D	Kanalnummer (Kortval)
	+4	04	Klusterstorlek
(HD)	+5	24	Kanalnummer (Kortval)
	+6	20	Klusterstorlek
Ej använd	+7	25	Kanalnummer (Kortval)
	+8	01	Klusterstorlek

Enheten HD (Hard Disc) kan ej användas. Endast de två första enheterna är accessbara.

UFD DOS

Följande enhetstabell läggs upp:

Enhet	Adress	Default (hex)	Namn
DR	+0	04-1C	DR select
	+1		Ej använd
	+2	44 (D)	
	+3	52 (R)	
HD	+4	24	Kanal
	+5	20	Klusterstorlek
	+6	48 (H)	
	+7	44 (D)	
MF	+8	2C	Kanalnummer (kortval)
	+9	04	Klusterstorlek
	+10	4D (M)	
	+11	46 (F)	
MO	+12	2D	Kanalnummer (kortval)
	+13	01	Klusterstorlek
	+14	4D (M)	
	+15	4F (O)	
SF	+16	2E	Kanalnummer (kortval)
	+17	04	Klusterstorlek
	+18	43 (S)	
	+19	46 (F)	
	+20		Adress+20 - Adress+27 är
	+21		lediga
	+22		
	+23		
	+24		
	+25		
	+26		
	+27		
RM	+28		Kanal
	+29		Klusterstorlek
	+30	52 (R)	
	+31	4D (M)	

DR select talar om vilken enhet som skall accessas under DRx:. Här skall offseten ligga till aktuell enhet. UFD kodas 1EH.

Kanal ska kodas på följande sätt:

```
Bit  7 6 5 4 3 2 1 0

     x x y y y y y y

xx     = Controllertyp  0 - Winchester
                        1 - ABC 832/ABC 838 och DD88
                        2 - ABC 830
                        3 -

yyyyyy = Kanalnummer 0-63
```

UFD

En nyhet i DOS 6.XX är att UFD (User File Directory) kan användas. Normalt på en enhet finns ett bibliotek (directory) med plats för 256 filer. 256 filer är för lite på enheter typ Winchester. För att komma förbi detta kan nya bibliotek skapas. Ett UFD bibliotek är en vanlig fil med följande skillnader:

Namnet har små bokstäver i extension-delen, tex EGET.Ufd.

Filen kan ej läsas med INPUT eller GET pga att biblioteksektorerna saknar den normala filinformationen, som måste finnas först i varje sektor.

UFD-biblioteket har selectkod 1EH.

På adress OFFF7H och tre byte framåt ligger informationen om aktuell UFD.

OFFF7H	Sektoroffset (l)
OFFF8H	Sektoroffset (h)
OFFF9H	Selektkod

5.2 Användbara subrutiner

I BASIC och DOS finns det rutiner som ligger på en absolut adress och som kan anropas i egna assemblerprogram.

Det finns även andra rutiner som ej ligger på en absolut adress och därför ej bör användas då det inte är säkert att de ligger på samma adress i kommande versioner av BASIC.

Här följer en kort beskrivning av de rutiner som är absolut-adresserade, och i slutet även några rutiner som ej är absolut-adresserade, men som ändå kan vara av intresse.

5.2.1 Subrutiner i BASIC

Adr(H)	Adr(D)	Namn	Förklaring
0000	0	HD.START	Initierar datorsystemet (Samma som att trycka på RESET-knappen). In: --- Ut: --- Utnyttjade register: ---
0002	2	CONSI	Läser ett tecken från tangentbordet (GET). Ekar ej till skärm. In IX = OFF4CH Ut A = inläst tecken Utnyttjade register: AF
0005	5	CONREAD	Läser en rad från tangentbordet (tills RETURN) (INPUT LINE). In HL = Buffertpekare BC = Max. buffertlängd Ut (HL) ->; Utnyttjade register: DE,AF
0008	8	BRKPNT	Reserverad för assembler-TRACE. Utför endast JP OFF9AH. In --- Ut --- Utnyttjade register: ---
000B	11	CONWRITE	Skriver en rad på bildskärmen (PRINT). Kan innehålla ESC-sekvens. In HL = Pekare till text BC = Längd Ut --- Utnyttjade register: ---

0010 16 RST.ERR

Skriver ut BASIC-felmeddelande.
Anropas med RST 10H.
DEFB errornr.
In ---
Ut ---
Utnyttjade register:

0012 18 SOFTNOCO

Samma som ovanstående fast errornr fås från register A.
In A = Felkod
Ut ---
Utnyttjade register:

0018 24 SKIPSP

Läser förbi blanktecken.
In HL = Pekare i text
Ut HL = Pekare till 1:a tecken skiljt från blank
Utnyttjade register:
AF

001B 27 STRCOMP

Stränguttrycksscanning. Används vid syntaxkontroll av instruktioner.
In DE = Internkodsbuffert
HL = Textbuffert
Ut (DE) -> Internkod
Utnyttjade register:
AF,HL

001D 30 TYPCOMP

Som ovanstående men B = typ.
In B = typ
ex 0 Flyttal
1 Heltal
2 Sträng
DE = Internkodsbuffert
HL = Textbuffert
Ut (DE) -> Internkod
Utnyttjade register:
AF,HL

0020 32 EVALU

Som RST28 men utan POP.
Ex. Lägg upp parameter på stacken.
Används bla av NAME- och KILL-rutinerna.

0023	35	LEXSCAN	Söker efter text (i DE). Texten skall separeras med en byte >= 80H. OFFH markerar slut på listan.
0028	40	RST28	Efter varje funktion (ex PEEK) görs RST 28H varefter återhoppsadr elimineras och stacken justeras.
002B	43	CHKLU	Testar om fil är öppen. In A = Filnr Ut IX = Filens ´Ix-map´ alt. errormeddelande ´32´
0030	48	RST30	RST 30H Gör ett JP OFF9DH.
0033	51	SOFTCONT	Anropar RESUME-rutin. Vid exekvering av RESUME görs återhopp till anropande rutin.
0036	54	NOTYET	Genererar BASIC-error 200.
0038	56	RST38	RST 38H Gör ett JP OFFAOH.
003B	59	HLBOFA	LD HL,BOFA.
003F	63	XBOFA	LD IX,BOFA.
0044	68	HLPROG	LD HL,BOFA+20.
0051	81	HLBUF1	Ladda HL med adr till inmatningsbuffert.
005A	90	IO	Anropar drivrutin som är knuten till IX-map. A = rutin IX = IX-map för fil (LU-block)
005D	93	OPNCMD	Öppnar fil. Gjord för kommandomod. A = Opentyp A = 0 Normalt OPEN A = 1 Normalt PREPARE A = 2 OPEN Sök först .BAC sedan .BAS A = 3 PREPARE Default .BAS 3 < A < 128 PREPARE Default .BAC A > 127 ISAM OPEN IX = Pekare till IX-map. HL = Pekare till filnamn.

0060	96	OPNX	Samma som ovan men namnet förutsätts vara formaterat.
0063	99	UNPFD	Kontroll och formatering av filnamn. HL = Pekare till oformaterat filnamn. DE = Buffert för formaterat filnamn. Från rutinen: NAME.EXT = OK.
0066	102	NMI	Adress för NMI (non-maskable interrupt) från BUS. Gör ett JP OFF8AH.
0069	105	USEUTLU	Stänger fil (temporär LU block på adr FF63H). OBS! Manipulerar stacken så att tvångsanrop kommer att göras.
006C	108	ALLOZMEM	Allokerar minne från HEAP och DE bytes (endast RUN TIME). Retur: TC=Carry satt=fick ej plats HL pekar på minnesarean DE=storlek Minnet nollställs (var II-BASIC).

5.2.2 Subrutiner i DOS

Adr(H)	Adr(D)	Namn	Förklaring
6000	24576	DOSINIT	Initierar DOS
6003	24579	RUNX	Exekverar ABS-fil A = Fysiskt filnr C = Fysiskt drivenr, 255 = alla drivar
6006	24582	LOADX	Laddning av ABS-fil A = Fysiskt filnr C = Fysiskt drivenr, 255 = alla drivar Returnerar: Carry = 0 --> HL = start adressen Carry = 1 --> A = felkod
6009	24585	SELROUT	Anropar en av fyra rutiner: A = Rutinnummer 1 PREPARE 2 CLOSE 3 OPEN 4 RESIZE
600C	24588	GETNC	Hämta nästa byte i DOSBUF0. Ökar buffertpekaren.
600F	24591	DR	Läs fysisk sektor till DOSBUF B/10H B = DOSBUFnr * 10H DE = Sektornr Returnerar: Carry = 1 --> False Zero = Drive off line. Carry = 0 --> True Zero = Paritetsfel
6012	24594	DW	Skriv fysisk sektor från DOSBUF B/10H B = DOSBUFnr * 10H DE = Sektornr Returnerar: Carry = 1 --> False Zero = Drive off line Carry = 0 --> True Zero = Paritetsfel
6015	24597	PREP	Preparera fil DE => Formaterat filnamn utan filpunkt Ex. BASICINISYS B = DOSBUFnr * 10H C = Selectkod
6018	24600	OPEN	Öppna fil DE => Formaterat filnamn utan filpunkt B = DOSBUFnr * 10H C = Selectkod
601B	24603	LOAD	Laddar en ABS-fil DE => Formaterat filnamn utan filpunkt B = DOSBUFnr * 10H C = Selectkod Returnerar: HL = Startadress

601E	24606	RUN	Laddar och anropar en ABS-fil DE => Formaterat filnamn uan filpunkt B = DOSBUFnr * 10H C = Selectkod
6021	24609	CLOSE	Stänger fil B = DOSBUFnr * 10H
6024	24612	CHOP	Tar bort utrymme på fil
6027	24615	PROTE	Gör direkt RET
602A	24618	POSIT	Random Access initiering DE = Logiskt sektornr B = DOSBUFnr * 10H
602D	24621	READ	Random Access, Läs logisk sektor DE = Logiskt sektornr B = DOSBUFnr * 10H
6030	24624	WRITE	Random Access, Skriv logisk sektor DE = Logiskt sektornr B = DOSBUFnr * 10H
6033	24627	GET	Läs nästa tecken ur buffert In B = DOSBUFnr * 10H Ut A = Tecken Carry = 1 Slut på block
6036	24630	GETR	Läs ett tecken med offset C ur buffert In B = DOSBUFnr * 10H C = Offset i buffert Ut A = Tecken Carry = 1 --> Slut på block
6039	24633	PUT	Skriv ett tecken sekvensiellt till buffert In A = Tecken B = DOSBUFnr * 10H Ut Carry = 1 Slut på block
603C	24636	PUTR	Skriv ett tecken till buffert med offset C In A = Tecken B = DOSBUFnr * 10H C = Offset i buffert Ut Carry = 1 Slut på block C = C +1
603F	24639	BSP	Sätt offsetposition i buffert till 0 (3) B = DOSBUFnr * 10H
6042	24642	BLKTF	Flytta ett block med data. C = Antal byte DE = Block flyttas till (Startadr) HL = Block flyttas från (Startadr)

6045	24645	TRAP	Initierar DOS ON ERROR In C = Felnummer DE = Hoppadress till felhanterare
6048	24648	RENAM	Byter namn på en öppen fil B = DOSBUFnr * 10H DE = Adress till filnamn (11 byte)
604B	24651		Otillåten. Ger oändlig loop
604E	24654		Otillåten. Ger oändlig loop
6051	24657		Otillåten. Ger oändlig loop
6054	24660		Otillåten. Ger oändlig loop
6057	24663		Otillåten. Ger oändlig loop
605A	24666		Otillåten. Ger oändlig loop
605D	24669	EXIT	Stänger alla filer+anropar CMDINT
6060	24672	DCWAI	Sätt drive, vänta till drive är klar Läs error/status
6063	24675	DW.0	Skriv sektor från DOSBUF0 DE = Fysiskt sektornr.
6066	24678	DR.0	Läs sektor till DOSBUF0 DE = Fysiskt sektornr

---

6069	24681	DEVDESP	Pekare till intern enhetslista i PROM. (UFD DOS/Winchester)
606B	24683	DEVDES	Pekare till intern enhetslista i RAM (UFD DOS/Winchester)
606D	24685	UFDENT	Pekare till UFD-entry på diskett (UFD DOS/Winchester)
606F	24687	DOSVER	DOS version nummer (UFD DOS/Winchester)
6070	24688	TYPE	DOS typ (UFD DOS/Winchester)
6071	24689	DRDWRET	Pekare till brytmöjlighet för läs-/skrivrutin (DR resp DW) (UFD DOS/Winchester)
6073	24691	CSS	Pekare till clustersize (UFD DOS/Winchester)
6075	24693	CHANN	Pekare till aktuell kortval byte (UFD DOS/Winchester)

FÖRKLARINGAR TILL BENÄMNINGAR I DOS-LISTA

DE => area	Register DE pekar på arean "area".	
DOSBUFnr	Kan vara 0,16,32,48,64,... (eller 00H - 70H) med steg 10H.	
Logiskt sektornr	0,1,2,...	
Fysiskt sektornr	0,1,2,...	På DOS ABC 6-1X (ABC830,DD80) skall dessa multipliceras med 32.
Selectkod	Normalt är:	0 = DR0: 1 = DR1: 2 = DR2: . . . . . . 255 = Sök på alla enheter
Fysiskt filnr	Bit 0 - bit 3	Bibliotekssektor
	Bit 4 - bit 7	Offset i bibliotekssektor

ÖVRIGT

Adr(H)	Adr(D)	Namn	Förklaring
7000	28672	OPTINIT	Initiering av options-PROM
7FFD	32765	HRLDIR	(HL) -> (DE) HL = Minne BC = Antal bytes Flyttning kan ske till och från HR-minnet.

5.3 Länkade listor

5.3.1 Funktionslista

För att BASIC-tolken skall kunna hitta de reserverade orden, som finns i BASIC II (inbyggda funktioner och instruktioner exempelvis INPUT, TAN etc) är orden placerade i diverse tabeller som är länkade inbördes.

I detta avsnitt skall vi behandla utseendet på funktionstabellerna och i nästa avsnitt behandlas tabellerna för instruktioner.

De tabeller som finns för funktioner är:

* Informationsblock

* Textlista

* Hopptabell vid exekvering (RUN-lista)

* Syntaxtabell

INFORMATIONSBLOCKET

Informationsblocket behövs för att länka flera listor. Detta för att möjliggöra en framtida utökning.

Utseendet på blocket är:

Adress	Vidarepekare(l)
Adress+1	Vidarepekare(h)
Adress+2	Offset
.	Funktionsantal
.	RUN-lista(l)
.	RUN-lista(h)
.	TEXT-lista(l)
.	TEXT-lista(h)
.	SYNTAX-lista(l)
Adress+9	SYNTAX-lista(h)

Vidarepekare Pekar till nästa funktionslistas informationsblock. Vill man behålla BASIC´s alla funktioner lägger man gamla adressen i dessa byte. Dvs

```
POKE adress,PEEK(65407),PEEK(65408)
```

Slut på listorna markeras med 0 (noll).

Offset Anger vilket värde som lagras som internkod enligt internkod=separatorbyte-128+Offset.

Funktionsantal Antal funktioner som finns i listorna.

RUN-lista Pekar till en vektor med adresser dit BASICen skall hoppa vid exekvering.

TEXT-lista Pekar till första byten i den lista som kallas LOOKUP-tabell vilken innehåller funktionernas namn lagrade som text.

SYNTAX-lista Pekar till en lista med funktionernas syntax bit-kodat.

TEXTLISTAN (LOOKUP)

Denna lista innehåller namnen på de reserverade orden.

Utseendet är:

1 byte	Separatorbyte
x byte	Text
1 byte	Separatorbyte
x byte	Text
.	.
.	.
.	.
1 byte	ETX-byte

Separatorbyte	Byte som separerar texterna från varandra. Innehåller även information om texten. Observera att den måste vara större än 128 (80 hex), dvs bit 7 = 1.
Text	Innehåller namnet på ett reserverat ord.
ETX-byte	Markerar att listan är slut. Är alltid satt till 255 (FF hex).

RUN-LISTAN

Anger till vilken adress BASIC skall hoppa då en funktion skall exekveras.

Utseendet är:

Adress	2 byte	Hoppadress funktion1
Adress+2	2 byte	Hoppadress funktion2
Adress+4	2 byte	Hoppadress funktion3
.	.	.
.	.	.
.	.	.

Hoppadress	Adress dit BASICen hoppar vid exekvering av funktionen, dvs JP "hoppadress".
	Adressen till hoppadressen beräknas ur:
	2*(Separatorbyte-128)+Adress

SYNTAXLISTAN

Anger vilken typ in- resp utparametrarna skall ha.

Utseendet är:

x byte	Separatorbyte
1 byte	Utparameter
x byte	Inparameter
x byte	Separatorbyte
1 byte	Utparameter
x byte	Inparameter
.	.
.	.
.	.

Separatorbyte	Motsvarar separatorbyten i TEXT-listan.
Inparameter	Innehåller information om variabeltyp på inparametern i bitformat.
Utparameter	Innehåller information om utdatas variabeltyp i bitformat.

IN- OCH UTPARAMETERBYTEN

Bit	7	6	5	4	3	2	1	0
	a	a	a	a	r	x	t	t

a a a a		t t	
0 0 0 0	Normal	0 0	Flyttal
0 0 0 1		0 1	Heltal
0 0 1 0	Kanske sista kod	1 0	Sträng
0 0 1 1		1 1	
0 1 0 0			
0 1 0 1		x	
0 1 1 0			
0 1 1 1	Variabelnamn	0	
1 0 0 0		1	
1 0 0 1			
1 0 1 0		r	
1 0 1 1			
1 1 0 0		0	Repetera ej koden
1 1 0 1		1	Repetera koden (x ggr)
1 1 1 0			
1 1 1 1			

5.3.2 Instruktionslista

Vi behandlar här bara skillnaderna mot uppläggningen för funktionerna, då utseendet på de bägge listorna i princip är lika.

Uppläggningen av informationsblocket skiljer sig bara på följande punkt: Adresserna 65405 och 65406 (FF7D resp. FF7E hex) är pekare till informationsblocket. I BASIC gör du alltså:

```
POKE länkadr,PEEK(65405),PEEK(65406)
POKE 65405,adress,SWAP%(adress)
```

där "adress" även nu är adressen till informationsblockets början och "länkadr" den pekare som pekar på nästa informationsblock i listan.

Syntaxlistan skiljer sig också från den som används vid funktioner.

SYNTAX-LISTA

Adress	2 byte	Hoppadress
Adress+2	2 byte	Hoppadress
Adress+4	2 byte	Hoppadress
.	.	.
.	.	.
.	.	.

Hoppadress — Adress dit BASIC hoppar för kontroll av eventuella parametrar till instruktionen. För varje ny instruktion måste även motsvarande syntaxkontroll skrivas i assembler. Dock kan många färdiga hjälprutiner anropas i BASIC. Se kap 5.2.1.

Adressen till hoppadressen beräknas ur:

2*(Separatorbyte-128)+adress

5.3.3 Utvidgning av BASIC

Vi skall här visa arbetsgången vid skapandet av egna funktioner och instruktioner som tillägg till BASICens. Fördelen med dessa är platsbesparing och ökad snabbhet.

ARBETSGÅNG

* Skapa rutiner i assembler som behandlar funktionen.

* Skapa en hopptabell dit BASIC skall hoppa vid exekvering.

* Skapa en lista som visar funktionens/instruktionens syntax.

* Lägga in namnen på de reserverade orden i en textlista.

* Initiera ett informationsblock för BASIC. Detta pga att BASIC söker efter funktioner enligt en länkad lista-metod. Blocket består till största delen av pekare till listorna ovan.

* Länka in informationen så att den blir tillgänglig för BASIC. Detta gör man genom att sätta adresserna 65407 och 65408 (FF7F resp. FF80 hex) resp adresserna 65405 och 65406 (FF7D resp FF7E hex) till början på informationsblocket för funktionen/instruktionen.

I BASIC skriver man:

```
POKE Flänkadr,PEEK(65407),PEEK(65408)
POKE Fiblock,Fadr,SWAP%(Fadr)
POKE Ilänkadr,PEEK(65405),PEEK(65406)
POKE Iiblock,Iadr,SWAP%(Iadr)
```

där

Fadr	är adressen till informationsblocket för funktioner.
Iadr	är adressen till informationsblocket för instruktioner.
Fiblock	är adressen till informationsblocket för funktioner.
Iiblock	är adressen till informationsblocket för instruktioner.
Flänkadr	är adressen till de byte som pekar på nästa informationsblock för funktioner.
Ilänkadr	är adressen till de byte som pekar på nästa informationsblock för instruktioner.

EXEMPEL

I nedanstående exempel kommer BASIC att utvidgas med två instruktioner och en funktion.

Instruktionerna är: PROCEDURE
WAIT

Funktionen är: GETITEM

PROCEDURE

Syntax: PROCEDURE FNfunktionsnamn

där "FNfunktionsnamn" är namnet på en funktion som finns definierad i programmet.

PROCEDURE används när programmeraren vill att en funktion enbart skall utföra satserna mellan DEF FNFunktionsnamn och FNEND. PROCEDURE returnerar ej värde. Jmfr Z=FNFunktionsnamn med PROCEDURE FNFunktionsnamn.

WAIT

Syntax: WAIT heltal

där "heltal" är antalet sekunder som programmet skall vänta.

WAIT medför att programmet ej utför några instruktioner det antal sekunder som "heltal" anger.

GETITEM

Syntax: GETITEM(itemsträng,itemnummer)

"Itemsträng" är en sträng med följande utseende:

CVT%¤(Antal)+CHR¤(Längd)+´Text´+ CHR¤(Längd)+´Text´+...

"Itemnummer" är numret på den deltext som skall returneras ur "itemsträng".

GETITEM returnerar en delsträng ur en itemsträng.

Exempel:
```
10000 A¤=CVT%¤(2)+CHR¤(3)+´Ett´+CHR¤(3)+´Två´
10010 ;GETITEM(A¤,1)
10020 ;GETITEM(A¤,2)
```

Detta ger utskriften:
```
  Ett
  Två
```

OBS!
Minsta värde för itemnummer är ett (1).