Commodore BASIC 2.0 second release

Commodore BASIC 2.0 second release för Commodore 64 och VIC-20

av Anders Hesselbom

Commodore BASIC 2.0 second release

Commodore BASIC 2.0 second release för Commodore 64 och VIC-20

av Anders Hesselbom

Publicerad 2021-07-22, revision 12, 2022-06-09. Rättningar och kommentarer skickas till anders@winsoft.se

Innehåll

KAPITEL 1: INTRODUKTION

Introduktion

Commodore 64 introducerades på marknaden år 1982 och erbjöd med sin tids mått mätt helt fantastiska möjligheter att skapa datoriserad animerad grafik och musik. Och även om prestandan på nya maskiner har ökat över tid, finns det ingen annan maskin än Commodore 64 som kombinerar sin nivå av enkelhet och tillgänglighet med samma imponerande grafik- och ljudkapacitet.

Commodore 64 levererades utan något riktigt operativsystem. En lekman som skulle kontrollera datorn erbjöds en ganska begränsad uppsättning av BASIC-kommandon. Den inbyggda BASIC-tolken, Commodore BASIC 2.0 second release, kunde även användas för att bygga enklare program, även om den saknade instruktioner för multimedia och många viktiga DOS-kommandon.

Commodore BASIC 2.0 second release var även det språk som fanns inbyggt i VIC-20. Commodore 128 hade ett rikare språk, Commodore BASIC 7.0, som är helt kompatibelt med Commodore BASIC 2.0 second release.

Den här boken vänder sig till dig som äger en VIC-20 eller en Commodore 64, som vill få ut det mesta ur datorns inbyggda kommandon. Eller till den som äger en Commodore 128 som vill använda den i Commodore 64-läge. För även om Commodore 128 avhandlas ytligt, är det just den BASIC som finns i VIC-20 och Commodore 64 som är i fokus.

Om du äger en VIC-20 eller en Commodore 64 så har du tillgång till en fantastisk maskin med två fördelar:

- Din dator är ytterst kapabel att utföra avancerade beräkningar, och beroende på modell, även utrustad med fantastiska grafik- och ljudmöjligheter.
- Din dator bjuder, till skillnad från många kontemporära maskiner, rimlig möjlighet att lära sig den till fullo. Hur den fungerar, hur den kontrolleras och hur man får ut det mesta ur den.

Att ha en dator som man bemästrar, gör att man är mindre beroende av andra människors mjukvara för att åstadkomma det man vill. Och kan man dessutom kommunicera direkt med mikroprocessorn, har man tillgång till en fantastisk beräkningskapacitet!

Så ha överseende med den tekniska introduktionen, för den här boken kommer att ge dig total kontroll över din Commodore 64, och det mesta är relevant även för en VIC-20 eller en Commodore 128. Sådant som innefattar minnesadresser, maskinkod och inbyggda funktioner skiljer de olika maskinerna åt.

Sådant som strikt handlar om språket Commodore BASIC 2.0 second release gäller för samtliga tidigare nämnda maskiner, samt i princip för Commodore 16, Commodore 116, Commodore Plus/4 och Commodore PET.

Detta kapitel tar upp bokens konventioner, en beskrivning av denna specifika BASIC-dialekt, denna specifika BASIC-dialekts olika versioner, språkets attribut, datorns textkonsol som används både för att skicka direkta kommandon och för att redigera ett datorprogram, datorns grafiska kapacitet och resten av bokens innehåll.

All källkod finns publicerad på GitHub, här:

https://github.com/Anders-H/CommodoreBASIC20/

Varje kodexempel i boken har en direktlänk med kodexempel.

Konventioner i boken

Indata som programrader eller kommandon skrivs med följande teckensnitt:

```
PRINT "HEJ"
```

Samma teckensnitt används för svaren från datorn.

Hänvisningar till tangenter på VIC-20 eller Commodore 64 skrivs med fetstil. Bilden visar till exempel **Return** till höger, **Run Stop** till vänster, och så vidare.

Figur 1: Tangentbordslayout på Commodore 64.

Den exakta tangentbordslayouten varierar beroende på vilken marknad du köpt din dator för. Bilden ovan visar en engelsk Commodore 64, men även en engelsk VIC-20 har **Return** och **Run Stop** placerad enligt bilden.

Ibland ska du trycka ner två tangenter. Om det står till exempel **Shift+A** ska **Shift** hållas nedtryckt medan **A** trycks ner.

Bildförklaringar och kodförklaringar skrivs i *kursiv stil*, som också används för att emfasera termer eller viktiga poänger. Även namn på felmeddelanden skrivs med kursiv stil.

Commodore BASIC 2.0 second release

Denna bok beskriver *Commodore BASIC 2.0 second release*, alltså det högnivåspråk från år 1980 som används i VIC-20/VIC-1001, Commodore 64 och Commodore 128 i 64-läge.

Många exempel fungerar även på Commodore 128 i sitt normala 128-läge, men kapitlen om grafik och ljud är inte relevanta i 128-läge eftersom den datorn har specifika kommandon för multimedia.

Boken beskriver inte hur du kopplar in din dator, eller hur den fungerar rent praktiskt. Det kan vara bra att veta innan, och den informationen finns i datorns manual.

Boken beskriver bara ytligt maskinkodsprogrammering och befintlig mjukvara till de olika datorerna, som en introduktion.

Vill man fördjupa sig i maskinkod för Commodore 64 rekommenderar jag boken "Machine Language for the Commodore 64, 128, and Other Commodore Computers" av **Jim Butterfield** (ISBN 978-0893036645).

För den som kör VIC-20 rekommenderar jag "VIC-20 Machine Code" av **Bruce Smith** (ISBN 978-0906812792).

Versioner

Commodore BASIC finns i de versioner som presenteras nedan.

Version 1.0 för Commodore PET 2001 som baseras på Microsoft BASIC.

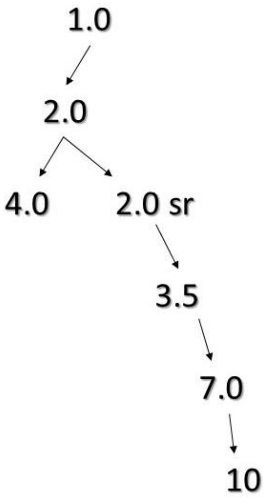
Version 2.0 för Commodore PET 2001 som är en vidareutveckling av version 1.0.

Version 4.0 för Commodore PET 4000 och CBM 8000 är den sista vidareutvecklingen av första version 2.0.

Version 2.0 second release som avhandlas i denna bok är buggrättad version av version 2.0 för PET 2001.

Version 4+ för Commodore CBM-II är en vidareutveckling på version 2.0 för PET 2001.

Version 3.5 för Commodore 16, Commodore 116 och Plus/4 är en vidareutveckling av Commodore BASIC 2.0 second release.



Figur 2: Språkets utveckling.

Version 7.0 för Commodore 128 är en vidareutveckling av version 3.5. Denna fanns även i prototypen Commodore LCD med versionsnumret 3.6

Version 10 utvecklades för prototypen Commodore 65. Varken Commodore 65 eller Commodore LCD nådde någonsin konsumentmarknaden. Tyska *MEGA Museum of Electronic Games and Art* arbetar med att få ut en färdigställd Commodore 65-klon på marknaden.

Språkets attribut

Commodore BASIC 2.0 second release är radnummerbaserat och innehåller 72 nyckelord för bland annat flödeskontroll, strängmanipulering och matematiska operationer.

39 av dessa är kommandon som utför något (CLOSE, CLR, CMD, CONT, DATA, DEF, DIM, END, FN, FOR, GET, GET#, GOSUB, GOTO, IF, INPUT, INPUT#, LET, LIST, LOAD, NEW, NEXT, ON, OPEN, POKE, PRINT, PRINT#, READ, REM, RESTORE, RETURN, RUN, SAVE, STEP, STOP, SYS, THEN, VERIFY och WAIT).

(Vissa kommandon kan förkortas. Som exempel kan LOAD skrivas som **L** och **Shift+O**. Förkortningarna beskrivs i din manual.)

24 av dessa är funktioner som eventuellt tar parametrar och ger ett värde tillbaka (ABS, ASC, ATN, CHR$, EXP, FRE, INT, LEFT$, LEN, LOG, MID$, PEEK, POS, RIGHT$, RND, SGN, SIN, SPC, SQR, STR$, TAB, TAN, USR och VAL).

Tre av dessa är logiska operatorer som tar en eller två operander och ger ett värde tillbaka (AND, NOT och OR). AND och OR är binära och tar alltså två operander, en på var sin sida, och NOT är unär och tar alltså tar en operand.

Det finns två nyckelord som inte passar in i någon särskild kategori (GO och TO), samt en inbyggd konstant (π). Slutligen finns tre inbyggda systemvariabler (STATUS, TIME och TIME$).

Commodore BASIC 2.0 second release har två lägen. Det ena kallas *direkt*, och innebär att man skriver en instruktion utan radnummer, som exekveras direkt när man trycker på **Return**. Det andra kallas *runtime*. Instruktioner som får ett radnummer, exekveras i runtime, alltså när programmet körs med (normalt) RUN. Om inget annat anges, kan alla kommandon användas både i direktläge och i runtime-läge.

De kodexempel som visas ska inte innehålla radbryte annat än i samband med ett nytt radnummer. Alla andra radbryten är gjorda av utrymmesskäl.

Alltså, följande kod ska skrivas på en rad, om inget annat sägs:

```
10 PRINT
   "HEJ"
```

Så här:

```
10 PRINT "HEJ"
```

Radbryten som orsakas av att terminalfönstret blir fullt har ingen effekt, det är radbryten som kommer av att man trycker på **Return** som har betydelse. Det går även att avgränsa programsatser med kolon, men detta bör undvikas[1]. Om du vill se koden utan felaktiga radbryten, finns en fotnot med en GitHub-länk till varje kodexempel.

De flesta exempel som visas i boken fungerar även i Commodore BASIC 7.0 på Commodore 128, men där finns ofta inbyggd funktionalitet som med fördel kan användas i stället. Commodore 128 beskrivs i ett eget kapitel. Tilläggen i Commodore BASIC 7.0 presenteras i appendix D och i appendix E presenteras en jämförelse mellan datorerna VIC-20, Commodore 64 och Commodore 128.

[1] När man bygger en `IF`-sats kan det vara nödvändigt att använda kolon, vilket förklaras under avsnittet om `IF` i kapitlet om kommandon.

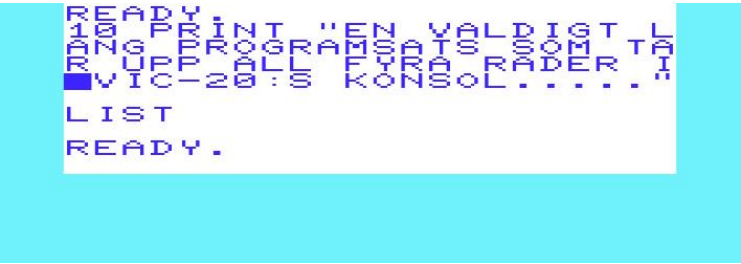
Om textkonsolen

VIC-20 har 22 kolumner per rad och Commodore 64 har 40 kolumner per rad. En programsats på VIC-20 består av max 88 (22×4) och en programsats på Commodore 64 består av max 80 (40×2) tecken. Om du på VIC-20 råkar skriva en programsats med radnummer som tar upp exakt fyra rader (eller två på Commodore 64) och sedan trycker **Return**, så lagras inte programsatsen. Datorn gör det felaktiga antagandet att du trycker **Return** på en tom rad.

Figur 3: En programsats på fyra rader på VIC-20 som inte blev lagrad.

(Ordet `READY` skrivs ut på skärmen när helst arbetet som tilldelats datorn är utfört, för att meddela att den är redo att ta emot nya instruktioner. Det sker oavsett om datorn har lyckats eller ej.)

För att raden faktiskt ska bli lagrad, måste du flytta textmarkören (med piltangenterna) upp till den sista raden som utgör programsatsen, och trycka **Return** där, som bilden nedan visar.



Figur 4: Placering av textmarkören för att ett tryck på Return ska lagra en fyra rader lång programsats på VIC-20.

Commodore 128 har inte detta problem.

Programsatser som skrivs med radnummer utförs inte, utan lagras i BASIC-minnet när man trycker **Return**. Programsatser som skrivs utan radnummer utförs direkt när man trycker **Return**. Vill man hoppa till nästa rad utan att lagra eller exekvera programsatsen man har skrivit, håller man nere **Shift** när man trycker på **Return**.

Notera att returtangenten, skifttangenten, med flera, är märkta på engelska även om du har en svensk dator med Å, Ä och Ö på tangentbordet. Så "skift" (som den normalt heter på svenska) benämns här som "shift", och så vidare.

För att summera:

Skriv ett kommando och tryck **Return** för att exekvera kommandot.

Skriv ett radnummer framför kommandot och tryck **Return** för att lagra kommandot i ditt aktuella program.

Tryck på **Shift**+**Return** för att gå till nästa rad utan att varken exekvera eller lagra något.

Bitmapsgrafik

VIC-20 kan växla mellan textkonsolen, högupplöst teckenbaserat grafikläge (176 * 184 pixlar) och flerfärgsgrafik (88 * 184 rektangulära pixlar), som bygger på block om 8 * 8 fysiska pixlar (eller 4 * 8 i flerfärgsläge - se avsnittet om flerfärg i kapitlet om grafik).

Commodore 64 och 128 (i 40-kolumnsläge) kan växla mellan textkonsolen, teckenbaserad grafik, högupplöst bitmapsgrafik (320 * 200 pixlar), flerfärgsgrafik (160 * 200 rektangulära pixlar) eller en kombination.

Commodore 128 i 80-kolumnsläge erbjuder inte bitmapsgrafik enligt dokumentationen. Commodore BASIC 2.0 second release innehåller inga kommandon för att komma åt vare sig bitmapsgrafiken eller datorns förmåga att spela ljud, men den här boken ger en kort introduktion i kapitlet om grafik.

För den som vill veta mer om grafik på Commodore 64 rekommenderar jag boken "Grafik och ljud på VIC 64" av **Sune Windisch** (ISBN 91-86398-17-2).

För VIC-20 finns boken "VIC-20 games, graphics and applications" av **David D. Busch** (ISBN 0-672-22189-6) som innehåller några enkla exempel som kan hjälpa dig igång.

Den här boken ger även en kort introduktion till ljud, trots att inte heller det täcks av Commodore BASIC 2.0 second release.

Bokens innehåll

Den här boken innehåller, förutom introduktionen, 16 kapitel och sex bilagor. Här följer en överblick över bokens kapitel, utöver detta första kapitel:

- Det andra kapitlet om **Datatyper** beskriver de olika formaten av data som kan lagras i minnet med Commodore BASIC 2.0 second release.
- Kapitlet **Kommandon** beskriver de instruktioner som utför något i språket.
- Kapitlet **Funktioner** beskriver de beräkningar som kan språket kan utföra genom att man anropar en inbyggd funktion.
- Kapitlet **Operatorer** beskriver de symboler eller ord som representerar en matematisk beräkning, som kan användas för att uttrycka en formel.
- Kapitlet **Övriga nyckelord** presenterar ytterligare några ord som är reserverade för inbyggd funktionalitet.
- Kapitlet **Inbyggda konstanter** presenterar ett namngivet värde som kan användas i språket.
- Kapitlet **Systemvariabler** beskriver de inbyggda variabler som kan läsas av i Commodore BASIC 2.0 second release.
- Kapitlet **Användardefinierade funktioner** visar metoder för att bygga subrutiner och egna funktioner.
- Kapitlet **16-bitarstal** förklarar hur man komponerar stora heltal som kan läsas av från inbyggda maskinkodsrutiner.
- Kapitlet **Metoder** presenterar några tillvägagångssätt för att lösa vanliga uppgifter i Commodore BASIC 2.0 second release.
- Kapitlet **Text** förklarar hur text och tecken kan manipuleras.
- Kapitlet **Grafik** ger en enkel introduktion till bitmapsgrafik. Om du äger en Commodore 64 får du även en introduktion till hur du använder datorns inbyggda sprites, något som bl.a. VIC-20 saknar.
- Kapitlet **Ljud** ger en enkel introduktion till datorns inbyggda synthesizer.
- Kapitlet **Felmeddelanden** beskriver vad som kan ha orsakat olika fel som kan uppstå i Commodore BASIC 2.0 second release.
- Kapitlet **Commodore BASIC 2.0 DOS** avhandlar filhantering som språket erbjuder. Detta kapitel vänder sig främst till dig som äger en diskdrive.

Boken har även sex bilagor.

Här följer en beskrivning av bokens sex bilagor kallade *appendix A-F*:

- **Appendix A** hanterar hur BASIC-program lagras på disk. Bilagan vänder sig till avancerade programmerare som vill utveckla verktyg.
- **Appendix B** listar de minnesadresser som refereras till i boken.
- **Appendix C** förklarar de termer som används i boken.
- **Appendix D** presenterar tilläggen i Commodore BASIC 7.0 jämfört med Commodore BASIC 2.0 second release.
- **Appendix E** jämför de tre datorerna VIC-20, Commodore 64 och Commodore 128.
- **Appendix F** ger en kort introduktion till maskinkodsprogrammering.

KAPITEL 2: DATATYPER

Datatyper

Variabelnamn består av ett eller två tecken (till exempel `X` eller `AB`) och eventuellt ett suffix som anger variabelns datatyp (procent eller dollar). När man refererar till en variabel kan hur många tecken som helst användas, men det är då endast de två första tecknen som identifierar variabeln. Eftersom en programsats i språket max består av 80 tecken, kan fler tecken en så aldrig vara aktuella. Det innebär att om du tilldelar värdet `20` till `ABC`, för att sedan läsa av värdet av till exempel `ABD` så kommer du att få `20` som svar. Följande kod skriver ut 20 på skärmen[2]:

```
10 ABC=20          Rad 10 tilldelar värdet 20 till variabeln AB*.
20 PRINT ABD       Rad 20 skriver ut värdet i variabeln AB* på skärmen.
```

Ett variabelnamn får inte innehålla något nyckelord, vilket innebär att till exempel `BEFORE` är ett ogiltigt namn, då det innehåller nyckelordet `FOR`. Nyckelord hittas alltså i en programsats även utan skiljetecken, vilket innebär att det inte spelar någon roll om det står `PRINT ABD` eller `PRINTABD` (eller `PRINTABC`) på rad 20 i programmet ovan.

Språket har stöd för tre datatyper. Dessa är *realtal*, *heltal* och *strängar*. Variablernas typ identifieras av ett suffix i variabelns namn.

Realtal saknar suffix, vilket innebär att följande kodrad lagrar realtalet `3.0` i variabeln `A`.

```
A=3
```

Realtalsvariabler använder sju bytes i minnet, där en håller reda på exponent, fyra är decimaldelen och två är variabelnamnet. Det innebär att realtalsvariabler kan hålla väldigt stora värden (från ungefär $2{,}9*10^{-39}$ till $1{,}7*10^{38}$).

Heltalsvariabler har ett procenttecken som suffix. Dessa variabler använder också sju bytes i minnet där två är värdet (som kan vara från `-32768` till `32767`), två är namnet. De resterande tre används inte till något. Exempel:

```
A%=23
```

[2] `https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/names.bas`

Tilldelning av ett för stort eller för litet värde ger felet *illegal quantity*, vilket är fallet i detta exempel:

```
A%=32800
```

Strängvariabler har ett dollartecken som suffix. Även dessa tar upp sju bytes. En för strängens längd, två för strängens minnesposition och två för namnet. De resterande två används inte till något. Utöver detta används en byte per tecken som strängen innehåller. Att bara en byte beskriver strängens längt säger oss att en sträng endast kan innehålla 255 tecken. Tecken lagras enligt PETSCII-tabellen. Följande program skapar en sträng bestående av 200 tecken, och skriver ut strängens längd på skärmen (alltså `200`)[3]:

```
10 FOR A=1 TO 100
20 A$=A$+"AA"
30 NEXT
40 PRINT LEN(A$)
```

Rad 10 påbörjar en iteration från 1 till 100.
Rad 20 lägger till värdet AA till vad som redan ligger i variabeln A$.
Rad 30 stänger iterationen.
Rad 40 skriver ut antalet tecken i variabeln A$, vilket är 200.

Om för många tecken försöker lagras i en strängvariabel uppstår felet *string too long*.

Två variabler av olika datatyp kan dela namn utan att någon konflikt uppstår. Det innebär att `A` kan innehålla ett värde samtidigt som `A%` innehåller ett annat och `A$` innehåller ett tredje.

Man kan även skapa matriser (arrayer) med kommandot `DIM`, vilket ökar antalet variabler som kan hanteras i BASIC-programmet markant, och dessutom bjuder på möjligheten att iterera över variabler, till exempel med `FOR`.

Datorns *call stack* (se ordförklaringar i appendix C) och alla variabler rensas av kommandot `CLR`, kommandot `NEW` och kommandot `RUN`.

[3] `https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/stringconcat.bas`

KAPITEL 3: KOMMANDON

Kommandon

Kapitlet tar upp de kommandon som språket har stöd för. Kommandon är generellt ord som utför något, som t.ex. utför ett programhopp, läs innehållet i en fil eller tar ett beslut. Dessa är `CLOSE`, `CLR`, `CMD`, `CONT`, `DATA`, `DEF`, `DIM`, `END`, `FN`, `FOR`, `GET`, `GET#`, `GOSUB`, `GOTO`, `IF`, `INPUT`, `INPUT#`, `LET`, `LIST`, `LOAD`, `NEW`, `NEXT`, `ON`, `OPEN`, `POKE`, `PRINT`, `PRINT#`, `READ`, `REM`, `RESTORE`, `RETURN`, `RUN`, `SAVE`, `STEP`, `STOP`, `SYS`, `THEN`, `VERIFY` och `WAIT`, och de avhandlas här i bokstavsordning.

CLOSE

`CLOSE` stänger filer eller enheter som öppnats med kommandot `OPEN`. Utelämnas `CLOSE` så kan filerna bli ofullständiga eller korrupta. `CLOSE` tar ett argument, och det är logiskt filnummer för filen som ska stängas (`0` till `255`). Det logiska filnumret bestäms när filen öppnas med `OPEN`. Om filnumret ligger utanför tillåten gräns inträffar felet *illegal quantity*.

Förutsatt att en skrivbar disk finns i enhet 8 och att den inte redan har en fil som heter `TEST`, skapar detta program en ny fil som heter just `TEST`, som innehåller texten `HEJ`[4].

```
10 OPEN
1,8,8,
"TEST, SEQ, W"
20 PRINT# 1,"HEJ"
30 CLOSE 1
```

Rad 10 öppnar en sekventiell fil på enhet 8 för, vilket normalt är diskdriven, för skrivning. Filen får enhetsnummer 1. Blankstegen i strängen är viktiga. Rad 20 skriver HEJ till filen (enhetsnummer 1). Rad 30 stänger filen.

Notera textsträngen på rad 10. Den berättar vad filen heter (`TEST`), vad det är för slags fil (`SEQ`) och vad som ska göras med den (`W`). Detta påverkar inte vad du kan göra med filen, men det gör att din diskdrive förstår vad för slags data som finns i filen, vilket har sina fördelar.

[4] `https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/close.bas`

CLR

`CLR` rensar minnet från alla variabler (inklusive arrayer) och användardefinierade funktioner. Dessutom nollställer kommandot `READ`-pekaren. Följande kod skriver ut `HEJ` en gång på skärmen, eftersom andra `PRINT`-satsen (rad 40) förekommer efter `CLR`-satsen (rad 30)[5].

```
10 A$="HEJ"
20 PRINT A$
30 CLR
40 PRINT A$
```

Rad 10 lagrar HEJ i strängvariabeln A$.
Rad 20 skriver ut innehållet i A$ (HEJ) på skärmen
Rad 30 rensar alla variabler.
Rad 40 skriver ut innehållet i A$, som nu är tom, på skärmen.

Följande exempel visar hur siffrorna `1`, `2` och `3` kan läsas två gånger, eftersom `CLR` nollställer `READ`-pekaren[6].

```
10 DATA 1,2,3
20 READ A:PRINT A
30 READ A:PRINT A
40 READ A:PRINT A
50 CLR
60 READ A:PRINT A
70 READ A:PRINT A
80 READ A:PRINT A
```

Rad 10 tillhandahåller konstanterna 1, 2 och 3.
Rad 20 läser in första konstanten till variabeln A och skriver ut dess värde (1).
Rad 30 läser in andra konstanten till variabeln A och skriver ut dess värde (2).
Rad 40 läser in tredje konstanten till variabeln A och skriver ut dess värde (3).
Rad 50 raderar alla variabler och nollställer READ-pekaren så att nästa konstant som blir läst är den första.
Rad 60 läser in första konstanten till variabeln A och skriver ut dess värde (1).
Rad 70 läser in andra konstanten till variabeln A och skriver ut dess värde (2).
Rad 80 läser in tredje konstanten till variabeln A och skriver ut dess värde (3).

Eftersom `READ`-pekaren nollställs på rad 50 blir resultatet `1`, `2`, `3`, `1`, `2` och `3`. Detta för att `CLR` förbereder datorn på att första värdet i första `DATA`-satsen är nästkommande värde som ska skickas till `READ`, alltså precis som när programmet startar första gången. Om du raderar rad 50, kommer rad 60 att resultera i felet *out of data*.

[5] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/clr1.bas

[6] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/clr2.bas

CMD

CMD styr om data som normalt går till skärmen, till en annan enhet. Detta skulle kunna vara en fil som öppnats med OPEN. Felet *file not open* uppstår om man försöker styra om data till en kanal som inte öppnats med OPEN. Kommandot PRINT#, följt av enhetsnummer, återställer skärmen som mottagare av data. Detta måste göras innan enheten stängs med CLOSE.

Förutsatt att en skrivbar disk finns i enhet 8 och att den inte redan har en fil som heter TEST, skapar detta program en ny fil som heter just TEST, som innehåller texten TILL FILEN. Dessutom skrivs ordet NORMAL ut på skärmen[7].

```
10 OPEN 1,8,8,
   "TEST, SEQ, W"
20 CMD 1
30 PRINT
   "TILL FILEN"
40 PRINT# 1
50 PRINT "NORMAL"
60 CLOSE 1
```

Rad 10 öppnar en sekventiell fil på disk (drive 8) som är heter TEST och är skrivbar.
Rad 20 dirigerar sådant som normalt går till skärmen, till den filen.
Rad 30 skriver text som blir dirigerat till filen efter att rad 20 körts.
Rad 40 återställer förändringen, så att skärmen återigen blir mottagare.
Rad 50 skriver text som inte blir dirigerat till filen, utan hamnar på skärmen.
Rad 60 stänger filen.

CMD går att använda i direktläge vilket är praktiskt om man till exempel vill skriva ut det BASIC-program man förtillfället har i minnet på printern. Normalt har printern enhet 4, men det kan variera. Följande exempel skrivs i direktläge och förutsätter att du har ett BASIC-program i minnet och att du har en påslagen skrivare med enhetsnummer 4[8].

```
OPEN 1,4
CMD 1
LIST
PRINT# 1
CLOSE 1
```

Öppnar enhet 4 (printer) som logisk fil 1.
Dirigerar sådant som normalt går till skärmen, till den logiska filen.
Skriver ut BASIC-programmet som för närvarande finns i minnet.
Återställer förändringen, så att skärmen återigen blir mottagare.
Stänger den logiska filen 1.

För mer information, se OPEN, PRINT#, LIST och CLOSE.

[7] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/cmd1.bas

[8] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/cmd2.bas

CONT

CONT fortsätter exekveringen av ett program som avbrutits, antingen av att kommandot END eller kommandot STOP påträffats i programmet, eller av att användaren tryckt på **Run Stop**. Om CONT anropas efter att programmet modifierats inträffar felet *can't continue*. Om CONT påträffas i ett program, fastnar programmet på den raden. Det finns alltså inget skäl att använda CONT i runtime.

Följande program skriver ut A på skärmen, och avslutas därefter. Om man återupptar exekveringen med CONT, skriver programmet ut B på skärmen[9].

```
10 PRINT "A"      Rad 10 skriver A på skärmen.
20 END            Rad 20 avbryter exekveringen.
30 PRINT "B"      Rad 30 skriver B på skärmen (om körningen återupptas med
                  CONT).
```

Om CONT används ytterligare en gång, har det ingen effekt.

Det går även att fortsätta programkörningen med RUN, om man anger önskad startrad som argument, men RUN rensar alla variabler, data-pekaren (se DATA och RESTORE) och hela datorns call stack.

Normalt använder man kommandot STOP för att bryta körningen i felsökningssyfte, för när STOP påträffas visas ett meddelande om att det har hänt: BREAK IN 20 (där 20 är radnumret där STOP påträffades).

Se även STOP och END.

DATA

DATA är ett kommando som används för att lagra konstant data i ett program. DATA listar helt enkelt värden som kan läsas av med kommandot READ. Kommandot utför egentligen inget, så om man hoppar till en DATA-sats med till exempel GOTO, fortsätter programmet att exekvera på den första efterföljande raden. Flera konstanter avgränsas med komma-tecken.

Commodore BASIC 2.0 second release har en pekare som håller reda på nästa värde som ska hämtas från DATA-satserna när READ används. Om READ används när alla värden redan är lästa eller om det över huvudet taget inte finns något värde att läsa, inträffar felet *out of data*.

[9] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/cont.bas

Följande program tillhandahåller tre heltalskonstanter. Två av dessa blir lästa in i variabler, och sedan utskrivna på skärmen[10].

```
10 DATA
   10,20,30
20 READ A%,B%
30 PRINT A% B%
```

Rad 10 tillhandahåller konstanterna 10, 20 och 30.
Rad 20 läser de två första konstanterna. Värdet 10 hamnar i A% och värdet 20 hamnar i B%.
Rad 30 skriver ut värdet av A% (10) och värdet av B% (20) på skärmen.

Resultatet av programmet blir att `10` och `20` skrivs ut på skärmen.

Om värdet som läses inte passar i variabeln som anges i `READ`-satsen på grund av typfel, till exempel en sträng i en heltalsvariabel, inträffar ett *syntax error*.

Maskinkodsprogram som distribueras som kodlistningar på papper brukar ibland levereras som `DATA`-satser i ett BASIC-program. Tanken är att då alla, utan extra mjukvara, kan skriva in ett BASIC-program i sin dator, som laddar in maskinkodsprogrammet i datorn (en så kallad *loader*). Betrakta följande program (endast Commodore 64)[11]:

```
10 FOR I=4096 TO 4101
20 READ A:POKE I,A
30 NEXT
40 DATA 169,5,141,33,208,96
```

Den som vill köra maskinkodsprogrammet som beskrivs på rad 40, skriver in hela detta BASIC-program och startar det med `RUN`. Det får maskinkodsprogrammet att lagras i minnet på position 4096, vilket sedan kan startas med:

```
SYS 4096
```

Har du gjort rätt, och använder du en Commodore 64, blir bakgrundsfärgen grön. Så vad gör maskinkodsprogrammet? Det vet du om du tar dig igenom appendix F om maskinkod.

Den som använder VIC-20 måste titta på en version anpassad för det systemet[12].

[10] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/data1.bas

[11] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/data2.c64.bas

[12] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/data3.vic20.bas

DEF

`DEF` används för att skapa numeriska funktioner som tar ett numeriskt argument och ger ett numeriskt värde. `DEF` följs av nyckelordet `FN` och därefter ska funktionens namn tillhandahållas. Funktionens namn följer samma regler som för numeriska variabler. Efter funktionsnamnet anges argumentets namn inom parentes, och slutligen efter ett likhetstecken anges funktionskroppen, som måste ge ett numeriskt värde. Alltså:

```
DEF FN FUNKTIONSNAMN(ARGUMENT)=FUNKTIONSKROPP
```

Detta exempel skapar en funktion som dubblar ett tal[13]:

```
10 DEF FN D(A)=A+A
20 PRINT FN D(10)
```

Rad 10 skapar funktionen A som dubblar värdet som skickas in.
Rad 20 använder kommandot FN för att anropa funktionen A.

Programmet resulterar i att värdet `20` skrivs ut på skärmen.

`DEF` kan inte användas i direktläge, utan endast i ett program (runtime-läge). Annars inträffar felet *illegal direct*. För mer information om `DEF`, se kapitlet om användardefinierade funktioner.

DIM

`DIM` används för att skapa en *array* med ett valfritt antal dimensioner (upp till 255 stycken). `DIM` tar variabelnamn och typ följt av arrayens sista index inom parentes. Följande kod skapar en flyttalsarray med 5 element (0 till 4):

```
DIM A(4)
```

Följande kod skapar en tvådimensionell heltalsarray med 5 x 5 element:

```
DIM A%(4,4)
```

Och följande skapar en tredimensionell strängarray med 7 x 7 x 7 element:

```
DIM A$(6,6,6)
```

Om man försöker använda element som inte skapats med `DIM` får man felet *bad subscript*. Om parenteserna lämnas tomma uppstår ett *syntax*-fel. Om man an försöker deklarera en array med fler än 32737 element uppstår felet

[13] `https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/def.bas`

illegal quantity. Om det inte finns ledigt minne för alla element (vilket är lätt hänt när många dimensioner används) uppstår felet *out of memory*.

Detta program skapar en *fibonaccisekvens* på 10 tal (med 1 som start) och skriver därefter ut dem[14].

```
10 DIM F%(9)          Rad 10 skapar en heltalsarray, F%, med 10 element (0 till 9).
20 A%=1               Rad 20 initierar A% med värdet 1.
30 B%=0               Rad 30 initierar B% med värdet 0.
40 FOR I=0 TO 9       Rad 40 påbörjar en iteration från 0 till 9.
50 F%(I)=A%+B%        Rad 50 lägger summan av A% och B% i ett element i arrayen F%.
60 A%=B%              Rad 60 kopierar värdet av B% till A%.
70 B%=F%(I)           Rad 70 kopierar värdet av aktuellt F%-element till B%.
80 NEXT               Rad 80 stänger iterationen.
90 FOR I=0 TO 9       Rad 90 påbörjar en iteration från 0 till 9.
100 PRINT F%(I)       Rad 100 skriver ut värdet ett värde från ett element i arrayen F%.
110 NEXT              Rad 110 stänger iterationen.
```

END

`END` avslutar ett BASIC-program, och är således bara användbart i runtime-läge, inte i direktläge. Om det finns fler programsatser efter att `END` har påträffats, kan `CONT` användas för att fortsätta exekveringen på raden efter.

Följande program skriver ut `A` på skärmen och avslutar. Genom att köra `CONT` i direktläge, fortsätter exekveringen och `B` skrivs ut. Därefter är programkörningen klar och programmet avslutas[15].

```
10 PRINT "A"          Rad 10 skriver ut A på skärmen.
20 END                Rad 20 avslutar programmet.
30 PRINT "B"          Rad 30 skriver ut B på skärmen om användaren väljer att fortsätta exekveringen.
```

Ett program behöver inte avslutas med `END`, eftersom det avslutas när exekveringen passerat sista programsatsen.

Rent tekniskt har `END` samma funktion som `STOP`, men `STOP` används normalt för att bryta programmet, till exempel för felsökning, medan `END` normalt används för att markera att programkörningen är slut, trots att det innehåller fler programsatser (till exempel subrutiner). Till skillnad från

[14] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/fibonacci.bas

[15] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/end.bas

`STOP` (som meddelar att programmet har brutits) ger `END` ett tyst avslut tillbaka till konsolen. `END` ska alltså användas för planerade programavslut.

FN

`FN` är ett nyckelord som anger att man vill göra ett anrop på en användardefinierad funktion, eller tillsammans med `DEF` om man vill skapa en användardefinierad funktion. Detta nyckelord är endast relevant i runtime-läge. `FN` beskrivs i kapitlet om användardefinierade funktioner, och där visas en nästlad funktion som skapats med `DEF` och anropas med `FN`.

Om en funktion anropas innan den definierats med `DEF` uppstår felet *undefined function* (skrivs *undef'd function*). Om funktionen är rekursiv uppstår felet *out of memory*. Om fler eller färre än ett argument används, uppstår ett *syntax error*.

FOR

`FOR` använder en numerisk variabel för att stega från ett värde till ett annat. Koden som skrivs mellan `FOR` och `NEXT` exekveras så många gånger det krävs för att nå målet – `NEXT` deklarerar var iterationen slutar. Följande exempel används variabeln `A` för att räkna från `1` till `5`. Det innebär att `HEJ` skrivs ut fem gånger på skärmen[16].

```
10 FOR A=1 TO 5
20 PRINT "HEJ"
30 NEXT
```

Räknar från 1 till 5.
För varje gång, skriv ut HEJ på skärmen.
Upprepa.

Underförstått ökar variabeln (i ovanstående exempel `A`) med 1 för varje gång. Skulle man skriva ut värdet av `A`, skulle man få `1` första gången, `2` andra, och så vidare. Sista gången har `A` värdet `5` och när iterationen körts klart, har `A` värdet `6`. Följande kod ger `HEJ 1`, `HEJ 2`, `HEJ 3`, `HEJ 4`, `HEJ 5` och `6`[17].

```
10 FOR A=1 TO 5
20 PRINT "HEJ"; A
30 NEXT
40 PRINT A
```

Räknar från 1 till 5.
För varje gång, skriv ut HEJ följt av värdet av A.
Upprepa.
Skriv ut värdet av A på skärmen.

[16] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/for1.bas

[17] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/for2.bas

Om man inte kan nå målet genom att öka angiven variabel med 1, exekveras innehållet i `FOR`-slingan (mellan `FOR` och `NEXT`) endast en gång. Mellan `FOR` och `NEXT` har variabeln då initialvärdet och efter initialvärdet + 1. Om rad `10` i ovanstående exempel ändras till `FOR A=7 TO 5` ger koden `HEJ 7` och `8` som svar.

Om man önskar räkna baklänges, eller stega med större eller mindre steg, kan steglängden anges efter `STEP` i samma programsats som `FOR`. I detta fall skrivs `HEJ` ut fyra gånger, och `A` har värdet `5.8` efter exekvering[18]:

```
10 FOR A=7 TO 6 STEP -0.3
20 PRINT "HEJ"; A
30 NEXT
40 PRINT A
```

Räkna från sju till sex, stega 0,3 steg bakåt.
Skriv HEJ följt av värdet av A
Upprepa.
Skriv ut värdet av A.

Programmet ger följande resultat:

```
HEJ 7
HEJ 6.7
HEJ 6.4
HEJ 6.1
 5.8
```

Iterationer kan nästlas med andra iterationer. I följande exempel räknar A från 1 till 10, och samtidigt räknar B från 1 till 10 för varje gång A räknar ett steg[19].

```
10 FOR A=1 TO 10
20 FOR B=1 TO 10
30 PRINT A;B
40 NEXT B
50 NEXT A
```

Rad 10 öppnar en iteration från 1 till 10.
Rad 20 öppnar en nästlad iteration, också från 1 till 10
Rad 30 skriver ut värdet av A (yttre iteration) och B (inre).
Rad 40 stänger den inre iterationen (B)
Rad 50 stänger den yttre iterationen (A).

[18] `https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/for3.bas`

[19] `https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/for4.bas`

Resultatet av körningen blir som följer:

```
1   1
1   2
1   3
1   4
1   5
1   6
1   7
1   8
1   9
1   10
2   1
2   2
2   3
2   4
2   5
2   6
2   7
2   8
2   9
2   10
3   1
3   2
3   3
3   4
```

...och så vidare upp till `10` och `10`.

Räknaren måste vara heltal eller flyttal, och en iteration kan endast sträcka sig från ungefär minus hundra miljarder till hundra miljarder, på grund av begränsningen i flyttalsvariabler, annars uppstår felet *overflow*. Dessutom kan endast nio `FOR`-slingor nästlas i varandra. Om en tionde skrivs in uppstår felet *out of memory*.

Om du undrar över hur blanksteg distribueras, läs kapitlet om text, avsnittet om `PRINT`.

En kommentar om prestanda: Att skapa iterationer med `FOR` ger bättre prestanda att skapa iterationer med `GOTO`, eftersom BASIC måste leta efter rätt radnummer varje gång `GOTO` används. En jämförelse presenteras på nästa sida. Om du är nyfiken på hur man mäter prestanda i Commodore BASIC, titta gärna på avsnittet om det i kapitlet om metoder.

Som prestandajämförelse, följande kod (endast Commodore 64) byter färg på bordern femhundra gånger. För att utföra uppgiften konsumerar datorn 721 jiffys[20].

```
10 PRINT "SPEED TEST: GOTO"
20 S=TI
30 X=0
40 C=0
50 X=X+1
60 C=C+1
70 IF X>255 THEN X=0
80 POKE 53280,X
90 IF C<500 THEN 50
100 PRINT TI-S
```

Samma funktionalitet, men utan `GOTO` och med `FOR` konsumerar endast 504 jiffys, vilket är anmärkningsvärt mycket snabbare[21]:

```
10 PRINT "SPEED TEST: FOR"
20 S=TI
30 X=0
40 FOR C=1 TO 500
50 X=X+1
60 IF X>255 THEN X=0
70 POKE 53280,X
80 NEXT C
90 PRINT TI-S
```

För att köra ovanstående jämförelse på VIC-20, ersätt `53280` mot `36879` i båda kodlistningarna. Resultatet på VIC-20 borde vara 618, respektive 435 jiffys.

[20] `https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/speedtest.goto.c64.bas`
[21] `https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/speedtest.for.c64.bas`

GET

Kommandot `GET` används i runtime för att läsa från tangentbordet. I direktläge ger kommandot felet *illegal direct*. `GET` tar en variabel av valfri typ, och när kommandot körs initieras variabeln med värdet från den senaste tangenttryckningen som står på tur.

`GET` väntar inte på att en tangent ska tryckas. När helst en tangent trycks ner under programkörningen lagras den i en buffert, och när helst kommandot `GET` körs hämtas nästa tecken från bufferten. (Undantaget vissa speciella situationer, till exempel när kommandot `INPUT` körs.) Om bufferten är tom initieras en numerisk variabel (till exempel `A` eller `A%`) med `0` och en sträng (till exempel `A$`) med en tom sträng. Bufferten rymmer tio tecken, så om 11 tecken har tryckts sedan `GET` anropades senast, är det bara de tio senaste av dem som kan läsas av. Flyttalsvariabler kan endast ta emot tecknen 0 till 9 och punkt (som ger värdet 0). Komma ger 0 och varningen *extra ignored* medan alla andra tecken ger felet *syntax*.

Detta innebär att man alltid bör använda `GET` tillsammans med en strängvariabel (suffixet `$`), och att man aldrig bör anta att strängvariabeln har ett värde. Kommandot `GET A$` kan alltså inte direkt efterföljas av `PRINT ASC(A$)`.

Följande exempel inväntar en tangenttryckning och skriver ut tangentens teckenkod på skärmen.

```
10 GET A$
20 IF A$="" THEN
     GOTO 10
30 PRINT ASC(A$)
40 GOTO 10
```

Rad 10 placerar värdet av nästa tangenttryck i bufferten i A$.
Rad 20 kontrollerar om bufferten var tom, och hoppar då tillbaka till rad 10 som försöker hämta på nytt.
Rad 30 skriver ut teckenvärdet för tangenten som trycktes.
Rad 40 inväntar nästa tangenttryckning.

För att förstå rad 30, läs gärna om `ASC` i kapitlet om funktioner.

Commodore 128 (Commodore BASIC 7.0) har ett kommando, `GETKEY`, som inväntar en tangenttryckning, vilket ungefär motsvarar ovanstående kods första två rader (`10` och `20`).

GET#

Commodore BASIC 2.0 second release kan hämta data från en inkopplad enhet, typiskt en diskdrive, men egentligen handlar det om vilken kompatibel enhet som helst. Kommandot `GET#` läser *ett tecken* från angivet filnummer och lagrar resultatet i angiven variabel.

Kommandot tar ett *logiskt filnummer* (se OPEN) och en lista över mottagande strängvariabler. Detta exempel (ett utdrag) läser ett tecken från den logiska filen 1 till variabeln X1$:

```
GET#1,X1$
```

Och detta exempel (igen, ett utdrag) läser två tecken från den logiska filen 6 till variablerna WA$ och WB$:

```
GET#6,WA$,WB$
```

Om den virtuella filen (6 i ovanstående exempel) inte är öppen inträffar felet *file not open*. Följande kod fungerar under förutsättning att du har en diskdrive inkopplad som enhet 8, att du har ett skivminne i den som har en fil som heter text.dat, innehållande texten HELLO WORLD.

```
10 OPEN 6,8,0,"TEXT.DAT"
20 FOR A=1 TO 11
30 GET#6,A$
40 PRINT A$;
50 NEXT A
60 CLOSE 6
```

Rad 10 öppnar en fil för läsning på enhet 6.
Rad 20 öppnar en sextonstegsiteration.
Rad 30 läser ett tecken i taget på enhet 6.
Rad 40 skriver ut tecknet utan radbryte.
Rad 50 stänger iterationen.
Rad 60 stänger filen.

Om alla förutsättningar är på plats, skriver detta ut HELLO WORLD på skärmen. Anledningen till att vi kör GET 11 gånger är att texten i filen är exakt 11 tecken lång (se kommandot STATUS). För mer information, se OPEN och även kapitlet om Commodore BASIC 2.0 DOS.

Om du har en diskdrive och en floppydisk med ledig plats, och vill testa ovanstående kod, kör först detta program:

```
10 OPEN 6,8,1,"TEXT.DAT"
20 PRINT#6,"HELLO WORLD"
30 CLOSE 6
```

Rad 10 öppnar en fil för skrivning på enhet 6.
Rad 20 skriver texten "HELLO WORLD till filen.
Rad 30 stänger filen.

GOSUB

GOSUB hoppar till angiven rad i ett BASIC-program, med möjligheten att återgå med RETURN. Om radnummer inte anges eller om raden som pekas ut inte finns inträffar felet *undefined statement* (skrivs *undef'd statement*). GOSUB förklaras i kapitlet om användardefinierade funktioner.

GOTO

`GOTO` används för att hoppa till önskad rad i ett BASIC-program. Om radnummer inte anges eller om raden som pekas ut inte finns inträffar felet *undefined statement* (skrivs *undef'd statement*).

Program som startas med `GOTO` behåller sina variabler från föregående exekvering, vilket illustreras av följande exempel:

```
10 A=A+1          Rad 10 ökar A med 1.
20 PRINT A        Rad 20 skriver ut värdet av A.
```

Varje gång programmet startas med `RUN` visas `1` på skärmen som resultat, men startas programmet med `GOTO 10` visas `1` första gången, `2` andra gången, och så vidare.

Följande program räknar upp från 1 till dess att användaren trycker **Run Stop**. Du borde alltså se `1`, `2`, `3`, och så vidare.

```
10 A=A+1          Rad 10 ökar A med 1.
20 PRINT A        Rad 20 skriver ut värdet av A.
30 GOTO 10        Rad 30 upprepar.
```

Men om du i stället ändrar rad 30 från `GOTO 10` till `RUN 10` så rensas alla variabler, och resultatet blir `1`, `1`, `1`, och så vidare. Igen, avbryt körningen genom att trycka **Run Stop**.

Kommandot `GOTO` kan även skrivas som `GO TO`, med ett blanksteg.

> `GOTO` bör användas för att skapa hopp i programmet. Villkorade hopp kan skapas om `GOTO` används tillsammans med `IF`. `GOTO` bör inte användas för upprepningar. För upprepningar ger `FOR` bättre prestanda.

IF

`IF` används för att förgrena programmet till att göra något om ett visst uttryck är sant, och eventuellt att göra något om samma uttryck är falskt. För att hoppa till ett specifikt radnummer används någon av följande meningsbyggnader:

```
IF [UTTRYCK] THEN GOTO [RADNUMMER]
```

...eller den nedkortade...

```
IF [UTTRYCK] THEN [RADNUMMER]
```

...eller...

```
IF [UTTRYCK] GOTO [RADNUMMER]
```

Det är också möjligt att utföra en eller flera programsatser om uttrycket utvärderas som sant. I detta fall skrivs `A` och `B` ut på skärmen:

```
10 A=10
20 IF A=10 THEN PRINT "A":PRINT "B"
```

Men i detta fall skrivs varken `A` eller `B` ut på skärmen:

```
10 A=5
20 IF A=10 THEN PRINT "A":PRINT "B"
```

Notera att alla programsatser efter `THEN` körs om uttrycket är sant, och att dessa programsatser ska vara avgränsade med kolon (`:`).

Ett uttryck (benämns `[UTTRYCK]` i exempelkoden ovan) måste utvärdera till sant (`0`) eller falskt (allt utom `0`). Dessa uttryck kan formuleras med jämförelseoperatorer (=, <>, <, >, <= eller >=). Jämförelseoperatorer ger `0` eller `-1` som svar, och `IF` anser att `0` är falskt och allt utom `0` (till exempel `-1`) är sant, vilket innebär att följande program skriver ordet `SANT` på skärmen...

```
10 IF 100 THEN PRINT "SANT"
```

Således, uttrycket `5=5` är sant, medan uttrycket `20<10` är falskt. För att kombinera flera jämförelser till ett uttryck som utvärderas till antingen sant eller falskt används de *logiska operatorerna*. Mer information om jämförelseoperatorer finns i kapitlet som avhandlar operatorer.

INPUT

Kommandot `INPUT` läser in data från tangentbordet och används för att samla in data från ditt programs användare. Data kan samlas in i samtliga format som Commodore BASIC 2.0 second release förstår, vilket är realtal, heltal och text. Som argument tar `INPUT` en variabel där inmatningen ska sparas, och det är den variabelns typ som avgör vad som kan skrivas av användaren. En felaktig inmatning, till exempel ordet `HEJ` till en numerisk variabel, till exempel `TZ%` (heltal) eller `AP` (realtal), ger felet *redo from start* till användaren, och därefter bes användaren försöka mata in ett värde igen.

Kommandot får inte användas i direktläge, endast i runtime-läge. Att rakt upp och ned skriva till exempel `INPUT A$` ger felet *illegal direct*.

"Prompten", alltså signalen för användaren att göra en inmatning med tangentbordet, är ett frågetecken. Det går att infoga tecken framför frågetecknet genom att tillhandahålla en sträng och ett semikolon framför variabeln som ska ta emot inmatningen, men frågetecknet visas alltid där.

```
10 INPUT "VAD HETER DU";A$
```

Ovanstående kodrad ger följande prompt:

```
VAD HETER DU?
```

Svaret lagras i variabeln `A$`. Följande program (nästa sida) kan användas för att testa inmatning, och vad som accepteras.

```
10 INPUT A
20 PRINT "OK!";A
30 INPUT A%
40 PRINT "OK!";A%
50 INPUT A$
60 PRINT "OK!";A$
```

Rad 10 samlar in ett realtal från användaren. Om ett heltal ges, accepteras det.
Rad 20 presenterar talet och körs först när ett realtal levererats.
Rad 30 samlar in ett heltal från användaren. Om ett realtal inom tillåtet omfång ges, accepteras det utan decimaler.
Rad 40 presenterar talet och körs först när ett tal har levererats.
Rad 50 samlar in en textsträng från användaren, oavsett om det ser ut som en text, ett heltal eller ett realtal.
Rad 60 presenterar den inmatade textsträngen.

Om användaren anger ett felaktigt värde, frågar `INPUT` igen. Därför bör man vara tydlig med vad som förväntas. Se bild på nästa sida.

Figur 5: INPUT säkerställer att användaren skriver in ett korrekt värde.

INPUT#

Kommandot `INPUT#` läser nästa tillgängliga post från angiven *logisk fil*, till angiven variabel. Kommandot beskriv utförligt i kapitlet om Commodore BASIC 2.0 DOS.

LET

`LET` används för att tilldela ett värde till en variabel. Efter `LET` skrivs variabeln som ska få ett värde, därefter värdet (eller ett uttryck som resulterar i ett värde). Värdet måste vara av en typ som kan lagras i variabeln (se kapitlet om datatyper) annars inträffar felet *type mismatch*. Följande program lagrar flyttalet 11,5 i `A`, heltalet 5 i `B%` och textsträngen HEJ i `C$`.

```
10 LET A=11.5
20 LET B%=5
30 LET C$="HEJ"
```

Det är frivilligt att uttryckligen skriva ut `LET`, vilket innebär att följande program ger exakt samma resultat:

```
10 A=11.5
20 B%=5
30 C$="HEJ"
```

LIST

`LIST` visar det BASIC-program som finns i minnet på skärmen (om inget annat har sagts). Man kan ange ett specifikt radnummer man vill se, eller ett omfång.

Skriv `LIST` för att se hela programmet (**Run Stop** avbryter listningen). För att titta på en specifik rad, skriv `LIST [Radnummer]`, till exempel `LIST 1020`. Man kan även lista alla programrader till och med ett visst radnummer genom att skriva `LIST -[RADNUMMER]`, programrader från och med ett visst radnummer genom att skriva `LIST [RADNUMMER]-` eller programrader mellan två radnummer, till exempel `100` och 200, genom att skriva `LIST 100-200`.

Så följande exempel visar alla programrader i BASIC-minnet vars radnummer är `500` eller högre:

```
LIST 500-
```

För att till exempel få ut programkoden på papper, se kommandot `CMD`.

LOAD

Kommandot `LOAD` hämtar ett BASIC-program eller någon annan data från någon lagringsenhet (till exempel dataset eller floppydisk).

Utan argument hämtar `LOAD` nästa fil (program eller annan data) från kassett (datasette).

Det första argumentet är filnamnet man vill hämta, vilket är obligatoriskt om man hämtar filen från floppydisk. Om man anger ett filnamn när man hämtar data från ett datasette, kommer din Commodore att skippa filer vars filnamn inte stämmer överens med namnet som anges.

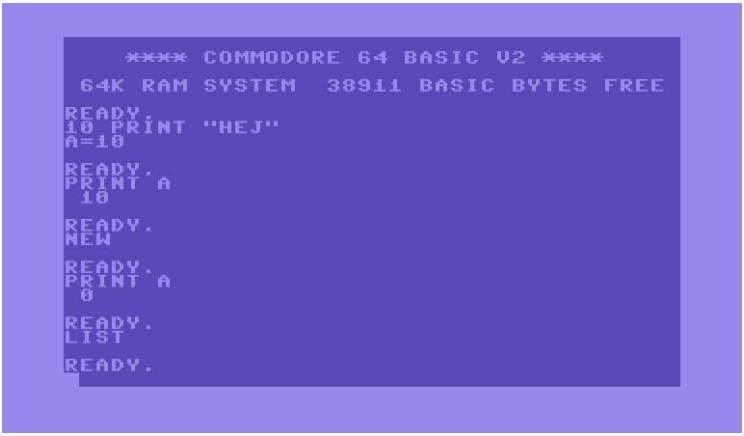
Det andra argumentet anger vilken enhet man vill hämta filen från. Om inget anges, antar din Commodore att det är enhet 1 som avses, vilket innebär datasette. 8 till 11 är typiskt diskdrives.

Det tredje argumentet, 0 eller 1, anger om filen ska laddas till BASIC-minnet (0) eller till den minnesadress som finns lagrad i filhuvudet (1), alltså filens två första bytes. Om argumentet utelämnas, antas BASIC-minnet vara den önskade destinationen.

För mer information, se kapitlet om Commodore BASIC 2.0 DOS.

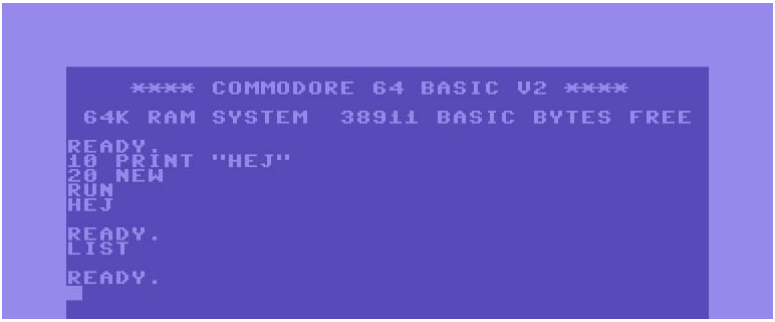
NEW

Kommandot `NEW` raderar BASIC-minnet och variabelminnet och ska användas när man vill få bort ett befintligt program ut minnet, och påbörja ett nytt.



Figur 6: Här visas hur NEW tömmer både variabel- och BASIC-minnet.

Om `NEW` förekommer i ett program, kommer programmet att raderas under körning.



Figur 7: Effekten av kommandot NEW i ett program.

NEXT

`NEXT` stänger en `FOR`-iteration. Följande exempel exekverar rad 20 fem gånger:

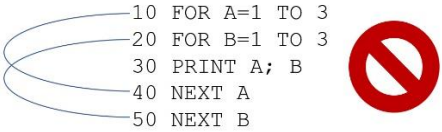
```
10 FOR A=1 TO 5
20 PRINT A
30 NEXT
```

Rad 10 räknar från 1 till 5.
Rad 20 skriver ut värdet av A (1, 2, 3, 4 och 5).
Rad 30 upprepar om uppräkningen ännu inte nått 5.

Koden ger `1`, `2`, `3`, `4` och `5` på skärmen.

Då `FOR`-iterationer kan nästlas (i max 9 nivåer) kan man även specificera vilken `FOR`-iteration man stänger. I ovanstående exempel antas `NEXT` stänga `FOR A`. Därför kan man på rad 30 skriva `NEXT` eller `NEXT A`. Skriver man något annat efter `NEXT` uppstår felet *next without for*.

Det finns fördelar med att faktiskt ange vilken `FOR`-iteration man stänger. till exempel kan BASIC-tolken avslöja eventuellt felaktigt nästlande, som till exempel denna:



Figur 8: Felaktigt nästlande.

En annan fördel är att antalet `NEXT`-satser kan kortas ner. I stället för att avsluta en korrekt nästlad iteration med två `NEXT`-satser, kan man lista de iterationer man avser att stänga enligt följande:

```
10 FOR A=1 TO 3
20 FOR B=1 TO 3
30 PRINT A; B
40 NEXT B, A
```

10: Räknar från 1 till 3.
20: För varje gång A räknas från 1 till 3 räknar B från 1 till 3.
30: Skriv ut värdet av A och värdet av B.
40 Upprepa A och upprepa B.

Programmet ger följande utdata:

```
1  1
1  2
1  3
2  1
2  2
2  3
```

```
3  1
3  2
3  3
```

Om fler än nio `FOR`-iterationer öppnas, uppstår felet *out of memory*.

ON

`ON` används tillsammans med `GOTO` eller `GOSUB` för att få programmet att hoppa till ett angivet radnummer baserat ett index. Kommandot byggs upp enligt följande:

```
ON Index GOTO|GOSUB Radnummer[, Radnummer…]
```

`Index` måste vara ett numeriskt värde. Om `Index` har värdet `1` kommer programmet att hoppa till det första angivna radnumret, och om `Index` har värdet 2 kommer programmet att hoppa till det andra angivna radnumret, och så vidare.

Notera gärna skillnaden mellan `GOTO` och `GOSUB`, som beskrivs både tidigare i detta kapitel och i kapitlet om användardefinierade funktioner.

Betrakta följande program:

```
10 A=1
20 IF A=1 GOTO 60
30 IF A=2 GOTO 80
40 IF A=3 GOTO 100
50 END
60 PRINT "1"
70 END
80 PRINT "2"
90 END
100 PRINT "3"
```

Det skulle kunna kortas ner till följande, tack vare `ON`:

```
10 A=1
20 ON A GOTO 40,60,80
30 END
40 PRINT "1"
50 END
60 PRINT "2"
70 END
80 PRINT "3"
```

Med tanke på hur variabler identifieras (se kapitlet om datatyper) kan ingen variabel i Commodore BASIC 2.0 second release heta något som börjar på `ON`. En variabel kan alltså inte heta `ONSDAG`.

Om `Index` är `0` eller om `Index` är ett positivt heltal som inte är representerat i den efterföljande listan av radnummer, fortsätter programmet på nästa rad, utan att göra något hopp.

Om `Index` inte är ett numeriskt uttryck inträffar felet *type mismatch*.

Om `Index` är ett negativt numeriskt uttryck inträffar felet *overflow*.

Om `Index` pekar ut ett radnummer som pekar på en icke-existerande programrad inträffar felet *undefined statement* (skrivs *undef'd statement*).

Men man måste tänka på att det finns fler situationer som kan orsaka flera typer av fel i runtime.

OPEN

Kommandot `OPEN` öppnar en *logisk fil* för skrivning eller läsning. En logisk fil skulle kunna vara en *fysisk fil*, en kanal till en skrivare, eller något annat. `OPEN` tar flera argument, där det första är obligatoriskt. Dessa är:

- Logiskt filnummer
- Enhetsnummer
- Sekundärt nummer
- Filnamn
    - Typ
    - Läge

Om typ och läge specificeras, sker det inom filnamnet.

Syftet med ett **logiskt filnummer** är att kommandon som opererar på en öppen kanal (`CLOSE`, `CMD`, `GET#`, `INPUT#` och `PRINT#`) ska kunna specificera vilken öppen kanal som avses. Din Commodore kan ha 10 logiska filer öppna samtidigt, och det logiska filnumret är ett heltal mellan 1 och 127.

Enhetsnumret anger vilken enhet du vill kommunicera med. Möjliga val är:

- **0**: Tangentbordet
- **1**: Kassettbandspelaren (som heter *datasette* enligt Commodores termologi) – om inget enhetsnummer anges antas 1
- **2**: Modem
- **3**: Skärmen
- **4-5**: Printer
- **8-15**: Diskdrive

Det **sekundära numrets** betydelse beror på enhet. Det skulle kunna vara val av teckenuppsättning för en skrivare eller val av operation på kassettbandspelaren (`1`). Det sekundära numrets betydelse för en diskdrive (`8` till `15`) beskrivs i kapitlet om Commodore BASIC 2.0 DOS, men det sekundära numret `15` betyder att man vill skicka ett kommando.

Filnamnet anges inom citattecken enligt: `"NAMN[, TYP[, LÄGE]]"`

Typ kan vara `SEQ`, `REL` eller `USR`. Läge kan vara `R` eller `W`.

Exempel: `"MINFIL, SEQ, W"`

Blankstegen ovan är viktig. Se kapitlet om Commodore BASIC 2.0 DOS för mer information.

POKE

En BASIC-programmerare skriver `POKE` ganska ofta när han jobbar på en Commodore-dator. Kommandot tar en minnesadress och ett heltal mellan `0` och `255` (en byte) och skriver helt enkelt heltalet till den angivna minnesadressen. Anledningen till att man vill använda `POKE` (och kanske även funktionen `PEEK` som läser på en angiven adress) i BASIC är inte att man vill lagra värden för sitt programs skull – för detta har vi variabler. Många av datorns funktioner, som till exempel att spela en ton eller att sätta en färg, kommer man åt genom att skriva direkt till datorns minne (se kapitlet om metoder). Ett annat motiv skulle kunna vara att lagra ett maskinkodsprogram i minnet (se appendix F). Att placera ut punkter (pixlar) i grafikminnet kan vara ytterligare ett motiv. Minnesadressen måste vara ett heltal mellan `0` och `65535`, men notera att vilket minne som är ledigt och vilken minnesadress som fyller vilken funktion varierar från dator till dator. En Commodore 64 och en VIC-20 delar inte samma minneskarta.

Om minnesadressen är mindre än `0` eller större än `65535`, eller om värdet som ska lagras är mindre än `0` eller större än `255` uppstår felet *illegal quantity*. Vilka adresser som kan användas fritt, vilka som fyller en funktion och vilka som är oanvändbara, beror på vilken maskin du använder.

Se även `PEEK`.

Tips: Det går snabbare att göra en `POKE` till en variabel adress än till en konstant adress.

En prestandajämförelse: Båda dessa program skriver ett värde till minnesadress `828` femhundra gånger. Denna variant konsumerar 214 jiffys på Commodore 64[22]:

```
10 T=TI
20 FOR A=1 TO 500
30 POKE 828,5
40 NEXT
50 PRINT TI-T
```

Medan denna endast konsumerar 128 jiffys på Commodore 64[23]:

```
10 T=TI
20 D=828
30 FOR A=1 TO 500
40 POKE D,5
50 NEXT
60 PRINT TI-T
```

På VIC-20 är tidsåtgången 180 respektive 108 jiffys.

PRINT

Kommandot `PRINT` skriver ut resultatet av ett eller flera uttryck, avgränsat med antingen kommatecken (för tabulator) eller semikolon. Efter det sista uttrycket läggs ett radbryte på, om man inte uttryckligen avslutar med semikolon eller komma. För mer information, se kapitlet om text.

[22] `https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/speedtest.poke.constant.bas`
[23] `https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/speedtest.poke.var.bas`

PRINT#

`PRINT#` skiljer sig från `PRINT` i att första argumentet ska vara ett logiskt filnummer (se både avsnittet om `OPEN` och kapitlet om Commodore BASIC 2.0 DOS för mer information). Det innebär att `PRINT#` kan användas för att skriva data till en logisk fil, som i sin tur kan vara till exempel en fysisk fil eller

Följande program förutsätter att du har en diskdrive inkopplad på enhet 8, att det finns lite ledigt utrymme på disketten som sitter i, att disketten inte är skrivskyddad och att disketten inte redan har en fil som heter `"text.dat"`.

```
10 OPEN 1,8,1,"HELLO.TXT"
20 PRINT#1,"HELLO WORLD!"
30 CLOSE 1
```

Rad 10 öppnar den fysiska filen HELLO.TXT för skrivning på kanal 1. Enheten antas vara 8 och det sekundära filnumret 1 indikerar skrivning.
Rad 20 skriver texten "HELLO WORLD!" till filen som är öppen på kanal 1.
Rad 30 stänger kanal 1.

Resultatet av denna körning blir en fil som heter `HELLO.TXT`, vars innehåll inte är något annat än följande sekvens av tecken (enligt PETSCII):

```
HELLO WORLD!
```

READ

`READ` läser nästa konstant från programmets `DATA`-satser, oavsett var i programmet de förekommer. `READ` tar en eller flera variabler som argument, och om antalet `READ`-anrop med tillhörande variabler överstiger antalet konstanter i `DATA`-satserna inträffar felet *out of data*. För att återställa, anropa `RESTORE`. Se avsnittet om `DATA` i detta kapitel för mer information.

REM

Kommentarer (alltså text som beskriver koden men som inte ska fylla någon funktion när programmet körs) kan skrivas på lite olika ställen. Det går att kommentera på skymda platser i programmet, som till exempel efter radnumret i en `GOTO`-sats eller en `GOSUB`-sats, som `HOPPA VIDARE` här:

```
10 GOTO 20 HOPPA VIDARE
20 END
```

Eller efter `END`:

```
10 GOTO 20
20 END:AVSLUTA
```

För platser som inte är skymda finns nyckelordet `REM` som tar ett frivilligt argument, en textsträng. Exempel:

```
10 PRINT "HEJ":REM "SKRIV HEJ"
```

Eftersom BASIC-tolken inte läser det som kommer efter `REM`, kan man strunta i att stänga strängen.

```
10 PRINT "HEJ":REM "SKRIV HEJ
```

Och om man inte skriver annat än icke-skiftade bokstäver och siffror, går det av samma anledning bra att helt strunta i citattecken.

```
10 PRINT "HEJ":REM SKRIV HEJ
```

Att helt undvika citattecken kan ge oväntade resultat, beroende på hur din Commodore-dator lagrar BASIC-program. För att se ett exempel där det blir fel, tryck ner **Commodore+Shift** på tangentbordet så att datorn växlar och visar gemener (se bild på nästa sida).

Figur 9: Tryck på Commodore+Shift för att växla mellan versaler och grafik eller gemener och versaler.

Du borde nu befinna dig i teckenläget för gemener och versaler och se gemener på skärmen, enligt bilden nedan. Om inte, tryck **Commodore+Shift** igen.

Figur 10: Läget för gemener och versaler.

I detta läge, skriv in följande program:

```
10 rem A B C
```

Skriv därefter `LIST` för att se vad som finns lagrat i BASIC-minnet. Du kommer att se detta:

```
10 rem atn peek len
```

Om du i stället skriver `rem"A B C` (alltså byter ut blanksteget mot ett citattecken) bevaras kommentaren så som den skrevs. För mer information om gemener och versaler, se kapitlet om text. Se tillägget om nyckelordskoder för mer information om hur BASIC-program lagras.

RESTORE

`RESTORE` nollställer pekaren som håller reda på hur mycket `DATA`-kommandot `READ` redan har läst, som visas i följande program:

```
10 DATA 1,2,3
20 READ A:PRINT A
30 READ A:PRINT A
40 RESTORE
50 READ A:PRINT A
```

Rad 10 definierar tre datakonstanter: 1, 2 och 3.
Rad 20 läser och skriver ut den första datakonstanten (1).
Rad 30 läser och skriver ut den andra datakonstanten (2).
Rad 40 återställer pekaren som håller reda på att två datakonstanter är lästa.
Rad 50 läser och skriver ut den första datakonstanten (1).

Resultatet av körningen blir:

```
 1
 2
 1
```

RETURN

`RETURN` återgår till raden efter den senaste `GOSUB`-satsen. Om `RETURN` anropas utan föregående `GOSUB`-sats inträffar felet *return without gosub*. Funktionaliteten illustreras av följande program:

```
10 PRINT "A"
20 GOSUB 50
30 PRINT "E"
40 END
50 PRINT "B"
60 GOSUB 90
70 PRINT "D"
80 RETURN
90 PRINT "C"
100 RETURN
```

Resultatet blir:

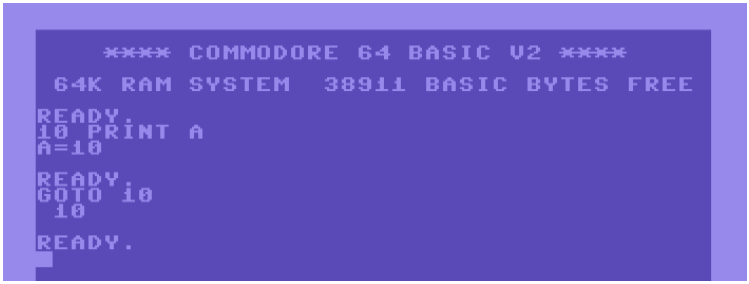
```
A
B
C
D
E
```

RUN

`RUN` rensar variabelminnet och startar det BASIC-program som finns i minnet, om något. Om ett radnummer anges, startar programmet på den raden. Om en rad som inte finns anges, uppstår felet *undefined statement*.

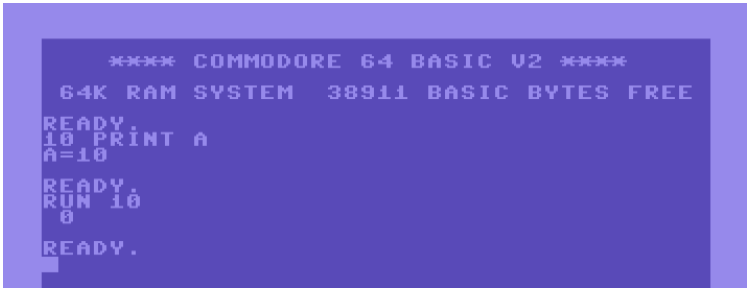
Det går att använda `RUN` i runtime för att göra hopp, vilket ger samma effekt som att kombinera `CLR` med `GOTO`.

Även `GOTO` kan användas för att starta program, men `GOTO` rensar inte variabelminnet så som `RUN` gör. Det illustreras här, där `A` fått ett värde, och när programmet som läser av `A` startas med `GOTO` behåller `A` sitt värde.



Figur 11: När ett program startas med GOTO bevaras alla variabler.

När programmet startas med `RUN` så rensas först alla variabler.



Figur 12: När ett program startas med RUN rensas alla variabler.

SAVE

`SAVE` sparar en anonym fil (till exempel ett program) på datasette eller en namngiven fil på antingen datasette eller på floppydisk. `SAVE` tar tre argument, som samtliga är frivilliga:

- Filnamn, om filen inte ska vara anonym
- Enhetsnummer
- Sekundärt nummer (0 eller 1)

Om enhetsnummer inte anges, antas `1` (datasette).

Diskdrives ligger på enhet `8` till `11`.

Det tredje argumentet, sekundärt nummer, har två olika betydelser.

För disk: Om sekundärt nummer inte anges, antas `0`, vilket innebär att BASIC-programmet i minnet ska sparas. `1` sparar annan data (till exempel ett maskinkodsprogram) från en absolut position, som då måste specificeras, vilket faktiskt inte fungerar Commodore BASIC 2.0 second release. Om du hamnar i situationen av att behöva spara specifikt minne, behöver du ett speciellt program för det. Se avsnittet om att spara minne på disk i kapitlet om metoder.

För kassettband: `0` har ingen funktion, men `1` skriver en slut-på-bandet-markering på bandet.

För mer information olika disk-operationer, se kapitlet om Commodore BASIC 2.0 DOS.

STEP

`STEP` används i iterationer som skapas med `FOR`, och anger hur stora steg en variabel ska öka eller minska i varje upprepning. `STEP` tar ett positivt eller negativt heltal eller flyttal. Om `STEP` utelämnas ökas iterator-variabeln med 1, så att skriva `STEP 1` har ingen funktion.

```
10 FOR A=1 TO 5 STEP 2
20 PRINT A
30 NEXT
```

Rad 10 räknar från 1 till 5 i variabeln A och ökar A med två varje gång.
Rad 20 skriver ut värdet av A.
Rad 30 upprepar iterationen.

När ovanstående kod exekveras blir rad 20 körd tre gånger. Programmets utdata blir:

```
1
3
5
```

Efter körningen har `A` värdet `7`. Om värdet efter `STEP` utelämnas inträffar ett *syntax error*. Om värdet efter `STEP` är `0` fastnar programmet i en oändlig iteration, som till exempel kan brytas av användaren med knappen **Run Stop** eller av kommandot `STOP`.

Eftersom FOR ökar iterator-variabeln med 1 när STEP inte används, kan följande kod:

```
10 FOR A=1 TO 5
20 A=A+1
30 PRINT A
40 NEXT
```

...kortas ner till följande:

```
10 FOR A=2 TO 6 STEP 2
20 PRINT A
30 NEXT
```

I båda fallen blir resultatet `2`, `4` och `6`.

STOP

`STOP` avbryter ett exekverande program, men bevarar alla variabler samt datorns call stack, så att programmet kan återupptas med kommandot `CONT`. `STOP` har samma effekt som om användaren trycker **Run Stop**, och har ingen effekt i direktläge, endast i runtime-läge.

```
10 A=10
20 PRINT A
30 STOP
40 PRINT A
```

Rad 10 lagrar värdet 10 i variabeln A.
Rad 20 skriver ut värdet av A (som är 10) på skärmen
Rad 30 bryter programmet.
Rad 40 exekveras inte, om inte kommando ges.

Om programmet startas med `RUN` skrivs `10` ut på skärmen, därefter bryts programmet med meddelandet `BREAK IN 30`. Om programmet återupptas med `CONT`, skrivs `10` ur på skärmen igen. Däremot, hade programmet återupptagits med kommandot `RUN 40` i stället, hade `0` skrivits ut på skärmen, eftersom `RUN` är ett av de kommandon som rensar alla variabler.

Rent tekniskt har `STOP` samma funktion som `END`, men `END` används normalt för att markera att programkörningen är slut, trots att det innehåller fler programsatser, medan `STOP` normalt används bryta programmet, till exempel för felsökning.

SYS

I Commodore BASIC 2.0 second release är `SYS` ett ganska enkelt kommando. Den tar en minnesadress, och exekverar maskinkodsprogrammet på den angivna adressen. När maskinkodsprogrammet avslutar (med operationskod 96 – `RTS`) eller kraschar, återgår datorn till BASIC-programmet.

`SYS` når både inbyggda rutiner i din dator, och dina egna maskinkodsrutiner. För mer information, se kapitlet om användardefinierade funktioner och appendix F om maskinkod.

THEN

`THEN` används tillsammans med `IF` för att ange vilket kommando som ska köras om uttrycket som står efter `IF` är sant. Om ett heltal anges efter `THEN` antas kommandot vara `GOTO`. För exempel, se `IF` tidigare i detta kapitel.

VERIFY

`VERIFY` kontrollerar att ett BASIC-program sparat med kommandot `SAVE`, är intakt, genom att jämföra programmet i minnet med vad som blev skrivet på kassett eller disk. Kommandot beskrivs i kapitlet om Commodore BASIC 2.0 DOS.

WAIT

Kommandot `WAIT` fryser programkörningen till dess att något visst kriterium är uppfyllt i datorns minne. WAIT tar två eller tre parametrar. Parametrarna är *minnesadress*, *mask 1* och *mask 2*. Följande beteende gäller om `WAIT` används med två parametrar:

BASIC-programmet fryses till dess att den angivna adressen har ett bitvärde som fullt ut matchar *mask 1*. Om värdet `6` lagras på adress `8192` enligt `POKE 8192,6` så kommer inte `WAIT 8192,4` att orsaka någon paus, eftersom `4` kontrollerar bit 2 (tredje biten från höger) och värdet `6` innebär att både bit 1 och 2 (andra och tredje biten från höger är satt).

Följande beteende gäller om `WAIT` används med 3 parametrar:

BASIC-programmet fryses till dess att den angivna adressen har ett bitvärde som matchar *mask 2* enligt `OR` och sedan till dess att den angivna adressen har ett bitvärde som full ut matchar *mask 1* (logiskt `AND`, som när två parametrar används).

`WAIT` används typiskt när ett BASIC-program samkörs med ett maskinkodsprogram som utnyttjar *interrupten*, men det finns speciella fall där `WAIT` kan användas i BASIC-program, till exempel när *multitasking* används.

Vilka kommandon rensar minnet?

Denna tabell är en sammanställning av vilka kommandon som raderar vad ur minnet. Kommandon visas horisontellt, de olika delarna i minnet visas vertikalt.

	RUN	NEW	CLR
Programkoden		✓	
READ-pekaren	✓	✓	✓
Variabler	✓	✓	✓
Callstacken	✓	✓	✓
Användardefinierade funktioner	✓	✓	✓

Programkoden är det som utgör ditt BASIC-program, alltså alla programsatser i minnet som har radnummer.

READ-pekaren förklaras i avsnittet om CLR i detta kapitel.

Variablerna är allt ditt namngivet data i minnet.

Callstacken förklaras i kapitlet om ordförklaringar.

Användardefinierade funktioner beskrivs i kapitlet om användardefinierade funktioner i avsnittet om DEF och i avsnittet om FN.

KAPITEL 4: FUNKTIONER

Funktioner

Commodore BASIC 2.0 second release levereras med 24 inbyggda funktioner. Generellt ger funktioner ett numeriskt värde som svar. De funktioner som ger en sträng som svar, har ett namn som slutar med ett dollartecken, till exempel `MID$`. Funktionerna är `ABS`, `ASC`, `ATN`, `CHR$`, `EXP`, `FRE`, `INT`, `LEFT$`, `LEN`, `LOG`, `MID$`, `PEEK`, `POS`, `RIGHT$`, `RND`, `SGN`, `SIN`, `SPC`, `SQR`, `STR$`, `TAB`, `TAN`, `USR` och `VAL` och de avhandlas här i bokstavsordning, men skulle kunna delas upp i följande X kategorier:

- Matematiska funktioner (9 stycken): `ABS`, `ATN`, `EXP`, `LOG`, `RND`, `SGN`, `SIN`, `SQR` och `TAN`
- Typomvandling och konvertering (5 stycken): `ASC`, `CHR$`, `INT`, `STR$` och `VAL`
- Systemfunktioner (2 stycken): `FRE` och `PEEK`
- Funktioner för manipulation eller analys av text (4 stycken): `LEFT$`, `LEN`, `MID$` och `RIGHT$`
- Funktioner för layout (3 stycken): `POS`, `SPC` och `TAB`
- Programmeringstekniska funktioner (1 stycken): `USR`

ABS

`ABS` tar ett numeriskt uttryck och ger dess absoluta värde som svar. Det absoluta värdet är ett positivt tal som anger värdets avstånd från 0. Exempel: `ABS(1.1)` ger `1.1` som svar och `ABS(-1.1)` ger också `1.1` som svar.

```
PRINT ABS(-1.1)
```

Om parametern inte är numerisk ger funktionen felet *type mismatch* och om uttrycket är för stort eller för litet ger funktionen felet *overflow*.

`ABS` kan användas för att räkna ut skillnaden mellan två tal, utan att du behöver veta vilket som är störst.

```
PRINT ABS(3-5)
```

Ovanstående ger svaret 2.

Låt säga att du vill att något ska hända om värdet i en variabel är tillräckligt nära 0, och att det acceptabla omfånget är -0,5 till 0,5. Det skulle kunna uttryckas så här:

```
10 A=0.3
20 IF A>=-0.5 AND
   A<=0.5 THEN PRINT "OK"
```

Rad 10 tilldelar värdet 0,3 till variabeln A.
Rad 20 skriver ut OK på skärmen eftersom A har ett värde som ligger mellan -0,5 och 0,5.

Men det skulle också kunna kortas ner till följande:

```
10 A=0.3
20 IF ABS(A)<= 0.5
   THEN PRINT "OK"
```

Rad 10 tilldelar värdet 0,3 till variabeln A.
Rad 20 skriver ut OK på skärmen eftersom A har ett värde som ligger mellan -0,5 och 0,5.

Detta är användbart till exempel om du har en hastighet i X-led i en variabel, och du vill veta hur hög hastigheten är, oavsett om objektet färdas åt vänster eller höger.

ASC

`ASC` tar en sträng och ger första tecknets kod i PETSCII-tabellen. Utan argument ger funktionen ett *syntax error* och med en tom sträng ger funktionen felet *illegal quantity*. Detta exempel ger `65` som svar, eftersom `A` har värde 65 i PETSCII-tabellen:

```
PRINT ASC("A")
```

Följande program skriver ut koderna för alla tecken i en sträng. Eftersom strängen innehåller `A`, `B` och `C` blir resultatet `65`, `66` och `67`:

```
10 A$="ABC"
20 FOR I=1 TO LEN(A$)
30 PRINT
   ASC(MID$(A$,I,1))
40 NEXT
```

Rad 10 lagrar ABC i A$.
Rad 20 påbörjar en iteration från 1 till längden av värdet i A$ (3).
Rad 30 skriver ut PETSCII-värdet för ett tecken i taget i A$.
Rad 40 stänger iterationen.

Ett tänkbart användningsområde för `ASC` är att lagra text direkt i RAM-minnet. Följande kod lagrar texten `COMMODORE 64` på adress 8192.

```
10 A$="COMMODORE 64"
20 FOR I=0 TO LEN(A$)-1
30 POKE 8192+I,
   ASC(MID$(A$,I+1,1))
40 NEXT
```

Rad 10 lagrar COMMODORE 64 I A$.
Rad 20 påbörjar en iteration från 0 till länden av värdet i A$ - 1.
Rad 30 lagrar PETSCII-värdet för ett tecken i taget på adress 8192 och framåt.
Rad 40 stänger iterationen.

Funktionen `CHR$` konverterar en PETSCII-kod tillbaka till ett tecken.

ATN

`ATN` ger en kurvas tangent från ett numeriskt värde. Funktionen tar ett numeriskt värde (radian) som argument och ger ett numeriskt värde som svar. Om argumentet inte är numeriskt inträffar felet *type mismatch*, om värdet ligger utanför tillåtet omfång inträffar felet *overflow* och om det utelämnas inträffar ett *syntax error*.

Följande program ger tangenterna `0.785`, `1.107` och `1.249` från radianerna `1`, `2` och `3`:

```
10 FOR I=1 TO 3
20 PRINT ATN(I)
30 NEXT
```

Rad 10 påbörjar en iteration från 1 till 3.
Rad 20 skriver ut tangenten från radianen I (först 1, sedan 2 och sist 3).
Rad 30 stänger iterationen.

Följande program ger tangenterna `0.017`, `0.034` och `0.052` från graderna `1`, `2` och `3`:

```
10 FOR I=1 TO 3
20 PRINT ATN(I*π/180)
30 NEXT
```

Rad 10 påbörjar en iteration från 1 till 3.
Rad 20 skriver ut tangenten från graden I (1, 2 och 3).
Rad 30 stänger iterationen.

CHR$

`CHR$` omvandlar en PETSCII-kod (0 till 255) till ett tecken. Ett värde utanför 0 till 255 ger felet *illegal quantity*, ett icke-numeriskt värde ger felet *type mismatch* och utelämnandet av PETSCII-koden ger *syntax error*. Följande kod skriver ut A på skärmen:

```
PRINT CHR$(65)
```

Ett tänkbart användningsområde för `CHR$` är att läsa ut text som ligger lagrad i RAM-minnet. Följande kod lagrar först texten `INFOTROLL` på adress 8192, för att sedan skriva ut texten från minnet till skärmen.

```
10 A$="INFOTROLL"
20 FOR I=0 TO 8
30 POKE 8192+I, ASC(
   MID$(A$,I+1,1))
40 NEXT
50 PRINT
60 FOR I=0 TO 8
70 PRINT CHR$(
   PEEK(8192+I));
80 NEXT
```

Rad 10 lagrar värdet INFOTROLL I variabeln A$.
Rad 20 påbörjar en iteration från 0 till 8.
Rad 30 lagrar ett tecken från A$ i taget i minnet.
Rad 40 stänger iterationen.
Rad 50 skriver ut en blank rad på skärmen.
Rad 60 påbörjar en iteration från 0 till 8 igen.
Rad 70 läser ett tecken i minnet och skriver ut det på skärmen (utan att gå vidare till nästa rad).
Rad 80 stänger den andra iterationen.

Funktionen `ASC` konverterar ett tecken tillbaka till en PETSCII-kod.

COS

`COS` ger en projektion av x-axeln av en punkt på enhetscirkeln. `COS` tar ett argument som antas vara en radian. Följande exempel använder teckengrafik för att rita en kurva cosinuskurva på skärmen.

```
10 FOR Y=0 TO 40
20 X=INT(
   COS(Y/10)*19)+19
30 PRINT SPC(X) "*"
40 NEXT
```

Rad 10 påbörjar en iteration från 0 till 40.
Rad 20 räknar ut ett cosinusvärde och skalar det för teckengrafik.
Rad 30 använder resultatet från rad 20 för att göra en indragning, och skriver ut en asterisk.
Rad 40 stänger iterationen.

Ett icke-numeriskt värde ger felet *type mismatch*, ett för stor eller för litet värde ger felet *overflow* och utelämnat värde ger *syntax error*. Se även `SIN` och `TAN`.

EXP

`EXP` är en matematisk funktion som räknar ut exponenten av den matematiska konstanten `e`, vilket innebär att `EXP(1)` ger `e` (alltså Eulers tal) vilket här innebär `2.71828183`. Ett icke-numeriskt värde ger felet *type mismatch*, ett för stor eller för litet värde ger felet *overflow* och utelämnat värde ger *syntax error*. Se även `LOG`.

FRE

Funktionen `FRE` frigör minne som inte längre används, räknar ut det lediga minnets storlek och skickar tillbaka resultatet som ett heltal. På grund av hur heltalets bitmönster ser ut, kommer tal över 32767 att se ut som ett negativt tal. Det beror på att den binära representationen för heltal med värdet 32768 eller högre har en 1:a på den position som datorn använder för att hålla reda på om ett tal är negativt eller positivt. Detta kan man enkelt lösa genom att addera 65536 till talet om det är negativt. Detta program presenterar ledigt minne i din Commodore:

```
10 F=FRE(0)
20 IF F<0 THEN F=F+65536
30 PRINT F
```

Rad 10 frigör minne och sparar antalet tillgängliga bytes i F.
Rad 20 kontrollerar om problemet med negativ-flaggan är aktuellt, och om så, åtgärdar det.
Rad 30 skriver ut resultatet, vilket troligen är ungefär 38861 på Commodore 64 eller 3533 på en oexpanderad VIC-20. (Ett för lågt värde kan bero på ett hårdvarufel i datorn.)

Argumentet till `FRE` saknar betydelse, men det behövs för att BASIC-tolken ska förstå att det handlar om just ett funktionsanrop. Detsamma gäller för funktionen `POS`. Att anropa `FRE(0)` ger samma resultat som att anropa `FRE(1)`, men att anropa `FRE` utan argument ger felet *syntax*.

INT

`INT` är en funktion som konverterar ett uttryck, till exempel ett flyttal till ett heltal (se kapitlet om datatyper för mer information). Om `INT` anropas utan argument ges felet *syntax*. Om INT anropas med ett uttryck som inte kan översättas till ett heltal (till exempel en textsträng) ges felet *type mismatch*.

Om INT anropas med ett uttryck som ser ut som ett heltal men som är för stort eller för litet för din Commodore ges felet *overflow*.

LEFT$

`LEFT$` tar en textsträng och ett numeriskt antal mellan 0 och 255, och ger angivet antal tecken från strängen som svar. Om du tar de tre första bokstäverna från `HEJSAN`, så får du kvar `HEJ`. Det innebär att följande kod skriver ut ordet `HEJ` på skärmen:

```
PRINT LEFT$("HEJSAN",3)
```

Om fyra tecken begärs ut från en tre tecken lång sträng, skickas de tre tecken som finns tillgängliga tillbaka. Om `LEFT$` används med felaktiga argument uppstår felet *type mismatch*, och om argument saknas uppstår felet *syntax*.

Se även `MID$` och `RIGHT$`.

LEN

`LEN` ger antalet tecken i en sträng. Detta lagrar värdet `3` i variabeln `A`, eftersom textsträngen `HEJ` består av tre tecken:

```
A=LEN("HEJ")
```

Frånvaron av argument ger felet *syntax*, ett argument som inte är en sträng ger felet *type mismatch* och en för lång sträng (alltså en sträng längre än 255 tecken) ger felet *string too long*.

LOG

Funktionen `LOG` ger ett värde ur den naturliga logaritmen med basen *e* (Eulers tal). Det innebär att värdet av parameterns om skickas till `LOG` måste vara minst 1, annars uppstår felet *illegal quantity*.

MID$

`MID$` kan antingen användas för att läsa ut en del av en sträng eller för att läsa ut ett önskat antal tecken från höger (precis som `RIGHT$`). `MID$` finns alltså i två utföranden:

- Om `MID$` används med två argument (en text och ett antal) så är funktionen en synonym till `RIGHT$`
- Om `MID$` används med tre argument, läser den ut innehållet ur en sträng som finns på angiven start med angiven längd

Det innebär att följande kod skriver ut `EMARI` på skärmen, eftersom det är vad som finns på position fyra och fem tecken framåt i ordet `ANNEMARIE`:

```
PRINT MID$("ANNEMARIE",4,5)
```

Om någon parameter har fel typ inträffar felet *type mismatch*. Om urvalet inte träffar strängen som skickas in, skickas en tom sträng tillbaka. Om urvalet inte kan träffa strängen som skickas in inträffar felet *illegal quantity*.

Se även `LEFT$` och `RIGHT$`.

PEEK

Funktionen PEEK tar en minnesadress och ger värdet av den byte som ligger lagrad på den angivna adressen. Vill du veta värdet av den byte som ligger lagrad på adress 800, kan du skriva följande:

```
PRINT PEEK(800)
```

Kommandot som används för att skriva direkt till minnet heter POKE, och det används till allt från att mata in maskinkodsinstruktioner, lagra data och för att komma åt din dators olika funktioner som aktiveras av att värden sätts på olika minnesadresser. PEEK läser oavsett vad adressen som skickas in träffar. Det kan vara ROM-minnet, det kan vara en adress som används av systemet, det kan vara skärmminnet, BASIC-minnet eller ett minne som förvaltas av ditt program. Om ett för litet tal (mindre än `0`) skickas in som adress, eller om ett för stort tal skickas in (`65536` eller) inträffar felet *illegal quantity*. Vilka adresser som kan användas fritt, vilka som fyller en funktion och vilka som är oanvändbara, beror på vilken maskin du använder.

Se även `POKE`.

POS

Funktionen `POS` behöver inga argument, men på grund av att BASIC-tolken inte förstår att en funktion utan argument verkligen är en funktion, så måste ett argument av valfri typ skickas in – detta gäller även för `FRE`. `POS` svarar med markörens position i x-led, 0-baserat. Om du skickar följande kommando till din dator, kommer `A` att få värdet `4`.

```
PRINT "1234";:A=POS(0)
```

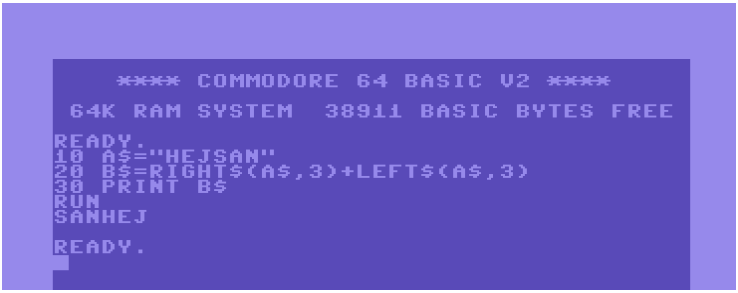
Du kan testa genom att skriva `PRINT A` i direkt-läge. Din dator svarar med att skriva ut `4` på skärmen.

Om du undviker semikolon (`;`) efter `"1234"` kommer `PRINT` att avsluta med ett radbryte, vilket innebär att `A` i stället får värdet `0`.

RIGHT$

`RIGHT$` fungerar på samma sätt som `LEFT$`, men `RIGHT$` plockar tecken från den inskickade strängens högra sida. Medan `PRINT LEFT$("HEJSAN",3)` ger svaret `HEJ` så ger `PRINT RIGHT$("HEJSAN",3)` svaret `SAN`.

Se även `LEFT$` och `MID$`.



Figur 13: Ett program som bygger en sträng av de tre sista och de tre första tecknen i en befintlig sträng.

RND

Funktionen `RND` ger ett pseudoslumptal mellan `0` och `1`. `0` är det lägsta möjliga svaret från `RND`, men `1` är över det högsta möjliga svaret. `RND` tar ett tal som argument, och talet avgör hur pseudoslumptalet ska räknas fram.

- Ett negativt tal väljer den talserie som talet representerar och ger `0` som svar
- `1` (eller annat positivt tal) tar nästa pseudoslumptal från den aktuella serien av tal
- `0` väljer någon av 60 olika serier (beroende på systemklockan) och ger det första talet som svar

Normalt anropas `RND` med ett negativt tal först, innan `RND` används tillsammans med `1` som argument, eller så anropas första inhämtningen av ett slumptal med `0` som argument och resterande framöver med `1` som argument.

För att skapa ett slumptal mellan `0` och `100`, använd `INT(RND(1)*101)`. Eftersom `RND` aldrig ger `1` som svar, kommer denna formel aldrig ge `101` som svar. För att skapa ett slumptal mellan `1` och `100`, använd `INT(RND(1)*100)+1`.

SGN

Funktionen `SGN` tar ett tal och svarar med talets signatur. Om talet som skickas in är positivt ger `SGN` talet `1` som svar. Om talet är negativt ger `SGN` svaret `-1` som svar. Annars `0`. En felaktig inmatning ger felet *type missmatch*. Följande program skriver ut ordet `NEGATIVT` på skärmen, eftersom `A` har värdet `-20`:

```
10 A=-20
20 B=SGN(A)
30 IF B=1 THEN PRINT "POSITIVT"
40 IF B=0 THEN PRINT "NOLL"
50 IF B=-1 THEN PRINT "NEGATIVT"
```

Rad 10 lagrar -20 i A.
Rad 20 lagrar information om huruvida A är positivt eller negativt i B.
Raderna 30-50 skriver ut om A är positivt eller negativt.

Om talet som skickas in är för stort eller för litet inträffar *felet overflow*, och om inget argument skickas in till `SGN` inträffar felet *syntax*.

SIN

`SIN` ger en projektion av y-axeln av en punkt på enhetscirkeln. `SIN` tar ett argument som antas vara en radian. Följande program ritar en ellips bestående av tecknet `*` på skärmen (endast Commodore 64):

```
10 FOR I=0 TO 63
20 X=INT(COS(I/10)*18)+19
30 Y=INT(SIN(I/10)*12)+12
40 POKE 780,0
50 POKE 782,X
60 POKE 781,Y
70 SYS 65520
80 PRINT "*"
90 NEXT
```

Rad 10 startar en iteration som ska köras 64 gånger.
Rad 20 och 30 hämtar nästa cosinus- och sinus-tal, och skalar upp det till lämplig storlek.
Rad 40-70 placerar textmarkören på positionen som hämtades på rad 20-30.
*Rad 80 skriver ut tecknet * på den platsen.*
Rad 90 upprepar (tillbaka till rad 20).

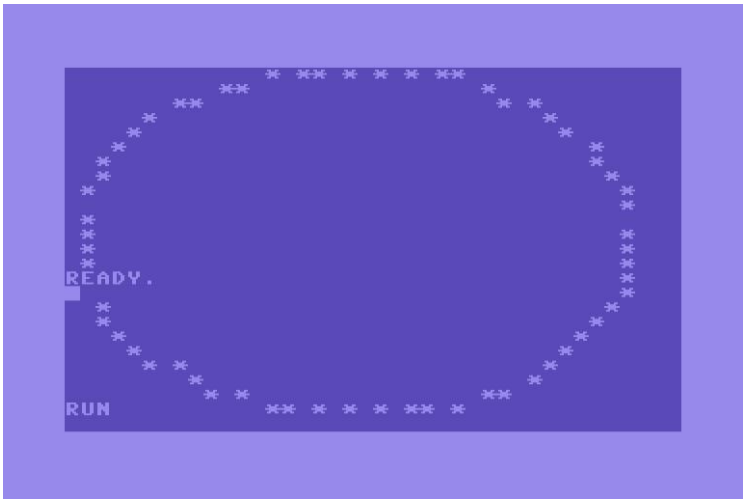
För att positionera textmarkören lagras `0`, värdet av `X` och värdet av `Y` i processorns register `A`, `Y` och `X` (rad 40-60). Därefter anropas en inbyggd rutin på adress `65520` (rad 70) som flyttar markören till den position som är angiven i processorns register.

För att få programmet att fungera på en VIC-20 måste X och Y få värden som är anpassade för en mindre textkonsol. Medan Commodore 64 kan visa

40 kolumner på 25 rader kan VIC-20 endast visa 22 kolumner på 23 rader. Om du vill köra programmet på en VIC-20, testa att ändra rad 20 och 30 till följande:

```
20 X=INT(COS(I/10)*10)+11
30 Y=INT(SIN(I/10)*11)+11
```

Så här ser resultatet av Commodore 64-versionen av programmet ut.



Figur 14: En cirkel skapad med COS och SIN på Commodore 64.

Ett icke-numeriskt värde ger felet *type mismatch*, ett för stor eller för litet värde ger felet *overflow* och utelämnat värde ger *syntax error*. Se även `COS` och `TAN`.

SPC

`SPC` ger ett angivet antal blanksteg. Följande kod...

```
PRINT "HELLO" SPC(4) "WORLD"
```

...ger följande svar:

```
HELLO    WORLD
```

Ett icke-numeriskt värde ger felet *type mismatch*, ett värde som är mindre än 0 eller större än 255 ger felet *overflow* och utelämnat värde ger *syntax error*.

SQR

`SQR` ger kvadratroten ur ett angivet tal. Kvadratroten av ett tal är det som multiplicerat med sig själv blir det angivna talet. Kvadratroten ur 49 är 7 eftersom 7 * 7 är 49.

STR$

Syftet med funktionen `STR$` är att producera en textrepresentation av ett numeriskt värde. Tanken är att man skickar in ett tal, t.ex. värdet av en heltalsvariabel, flyttalsvariabel, någon numerisk konstant eller något annat uttryck som utvärderas till ett tal, och får en läsbar textsträng som svar. Din dator är ganska snäll hantering av typer, åtminstone när ett tal ska representeras som text. Vi kan t.ex. skriva följande för att få en textrepresentation av talet nio:

```
PRINT 3*3
```

När konverteringen ska ske åt andra hållet, från text till tal, är reglerna betydligt mer strikta. Att försöka multiplicera `"3"` med `3` ger felet *type mismatch*. Men normalt så kommer ett numeriskt värde där ett strängvärde förväntas, göra samma sak som ett anrop på `STR$`, men inte alltid. Kommandot `PRINT 10` fungerar och kommandot `PRINT` levererar plikttroget en textrepresentation av 10, men om du tilldelar värdet 10 till en strängvariabel, uppstår återigen felet *type mismatch*, vilket `STR$` åtgärdar. Genom att säga att t.ex. `A$=STR$(10)` så har du deklarerat för interpretatorn att din typomvandling var avsiktlig, vilket gör att anropet släpps igenom.

Både uttrycklig konvertering med funktionen `STR$` och antydd konvertering med kommandot `PRINT` bjuder på en del formatering. En textrepresentation av ett numeriskt värde. 10^3 skrivs som `1E3`, vilket ger strängen `"1000"` som svar.

`STR$(10)` ger `"10"` som svar.

`STR$(1000)` ger `"1000"` som svar.

`STR$(1E3)` ger `"1000"` som svar.

TAB

Denna funktion används för att konfigurera nästkommande användningar av tabulatorn, vars storlek anges som argument. Givet att man skriver ut en sträng som är `X` tecken lång, kommer `TAB(Y)` att skapa `Y-X` blanksteg.

- `PRINT "SVEN" TAB(3) "B"` ger `SVENB`
- `PRINT "SVEN" TAB(4) "B"` ger `SVENB`
- `PRINT "SVEN" TAB(5) "B"` ger `SVEN B`
- `PRINT "SVEN" TAB(6) "B"` ger `SVEN  B`

Funktionen syftar till att bistå med formatering när mycket information ska presenteras på skärmen på ett strukturerat vis, med minskat beroende till föregående värde.

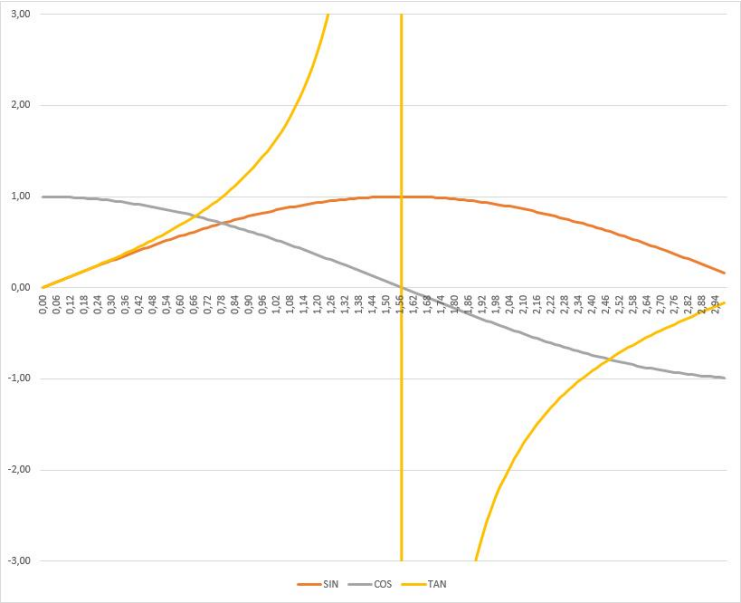
- `PRINT 1 TAB(5) "2"` ger `1    2` (med inledande blanksteg före `1`)
- `PRINT 11 TAB(5) "2"` ger `11  2`
- `PRINT 111 TAB(5) "2"` ger `111 2` (ett blanksteg mellan `111` och `2`)
- `PRINT 1111 TAB(5) "2"` ger `1111 2` (även här ett blanksteg mellan, vilket är så nära den avsedda platsen som möjligt)

TAN

Funktionen `TAN` räknar ut *tangenten* för en vinkel angiven som radianer (till skillnad från grader). En tangent är en rät linje som skär en linje, och funktionen `TAN` anger den vinkeln.

Du kan själv räkna ut tangenten genom formeln `SIN(n)/COS(n)` men du sparar lite minne och tolkningstid (några få bytes och jiffys) genom att istället anropa `TAN(n)`.

Om argumentet (benämns `n` ovan) saknas uppstår felet *syntax*. Om argumentet är av fel typ uppstår felet *type mismatch* och i vissa fall kan felet *division by zero* uppstå. Bilden på nästa sida visar vad `SIN`, `COS` and `TAN` ger för svar vid olika input.



Figur 15: Svar från funktionerna SIN, COS och TAN efter input

Se även `SIN` och `COS`.

USR

`USR` kan konfigureras att anropa en valfri maskinkodsfunktion. `USR` tar ett numeriskt värde och ger ett numeriskt värde som svar. `USR` måste alltid konfigureras i förväg för att en funktion ska bli anropad, och `USR` kan konfigureras till att ta ett strängvärde i stället för ett numeriskt värde. För mer information om `USR`, se kapitlet om användardefinierade funktioner.

VAL

Funktionen `VAL` söker efter numeriska värden i en sträng enligt några förutbestämda regler.

- `"45"` tolkas som det numeriska värdet `45`
- Blanksteg ignoreras så även `"  4  5 "` tolkas som `45`
- Icke-numeriska tecken ignoreras, men tolkningen slutar vid påträffande av icke-numeriska tecken, så `"4T"` tolkas som `4` men `"T4"` tolkas som `0`
- Inledande minus tolkas som negativt tal
- Inledande punkt tolkas som fraktion
- Argument som inte är textsträngar orsakar felet *type mismatch*
- Strängar som representerar för stora eller för små tal orsakar felet *overflow*

Exempel:

```
10 INPUT "SKRIV ETT TAL";A$
20 INPUT "SKRIV ETT TILL";B$
30 A=VAL(A$):B=VAL(B$)
40 PRINT "SUMMAN AV" A
   "OCH" B "ÄR" A+B
```

Rad 10 samlar in ett tal från användaren som sträng.
Rad 20 samlar in ett tal till i strängformat.
Rad 30 konverterar båda talen i strängformat till flyttalsformat. Felaktig inmatning ger programfel.
Rad 40 presenterar resultatet.

Resultatet av en körning kan se ut så här:

```
SKRIV ETT TAL? 4
SKRIV ETT TILL? 2
SUMMAN AV 4 OCH 2 ÄR 6
```

KAPITEL 5: OPERATORER

Operatorer

En operator är en symbol eller ett ord som representerar en matematisk operation. Commodore BASIC 2.0 second release erbjuder operatorer i två olika kategorier:

- *Jämförelseoperatorer*, som representeras av ett eller två tecken
- *Logiska operatorer*, som tar två värden som tolkas som *sant* eller *falskt* för att ge ett svar

Jämförelseoperatorer

Jämförelseoperatorer (=, <>, <, >, <= eller >=) tar två numeriska operander (heltal eller flyttal) eller två strängoperander och ger `-1` som svar uttrycket är sant, annars `0` om uttrycket är falskt. I ett program används jämförelseoperatorer typiskt i `IF`-satser. Jämförelseoperatorerna är följande:

Operator	Betydelse
=	Lika med. Tecknet används även för tilldelning, och det är sammanhanget som avgör betydelsen.
<>	Inte lika med, ger sant (`-1`) om operanderna är olika.
<	Mindre än.
>	Större än.
<=	Mindre än eller lika med.
>=	Större än eller lika med.

Följande kod skriver ut `-1` på skärmen eftersom 15 och 15.0 är värda lika mycket:

```
PRINT 15=15.0
```

Följande kod skriver ut `0` på skärmen eftersom strängen `ANDERS` inte alls kommer efter (är större än) strängen `STAFFAN`.

```
PRINT "ANDERS">"STAFFAN"
```

Logiska operatorer

Logiska operatorer (`AND`, `NOT` och `OR`) tar ett eller två uttryck som antingen utvärderas som sant eller falskt och ger sant (`0`) eller falskt (`-1` eller annat värde som inte är `0`) som svar. Logiska operatorer kan endas hantera numeriska operander, inte strängar. Commodore BASIC 2.0 second release har inga *booleska* variabler (sant eller falskt). De logiska operatorerna är *bitvisa* operatorer, vilket innebär att vid tester (som t.ex. kan uttryckas med

IF) antas 0 vara falskt icke-noll vara sant, samt att i beräkningar kommer en satt bit (1) vara sann och en icke satt bit (0) vara falsk. AND, OR och NOT kan alltså användas för binär aritmetik.

AND

AND är en logisk operator som tar två operander. Tillåtna operander är uttryck som utvärderas till heltal, flyttal eller till sant/falskt (inte 0 eller 0). AND används antingen i tester tillsammans med kommandot IF och ger då ett booleskt värde (0 eller -1) som svar, eller i bit-operationer och ger då ett heltal som svar.

Följande exempel visar hur AND används i tester med IF. Exemplet skriver inte ut något på skärmen, eftersom båda operander måste utvärderas som sanna för att operatorn AND ska ge sant som svar. Under alla andra förutsättningar ger AND falskt som svar.

```
10 A=1
20 B=2
30 IF A>1 AND B>1
   THEN PRINT "OK"
```

Rad 10 sparar värdet 1 i variabeln A.
Rad 20 sparar värdet 2 i variabeln B.
Rad 30 skriver OK på skärmen endast om både A och B har ett värde som är större än 1, vilket inte är fallet.

Detta exempel utför bit-operationen 3 AND 2 (alltså 00000011 AND 00000010) vilket ger svaret 2 (00000010).

```
10 PRINT 3 AND 2
```

Rad 10 skriver ut resultatet av 3 AND 2 på skärmen.

När AND används för bit-operationer tittar operatorn på samtliga bitar i de båda talen, och ger 0 som svar om inte båda talens bitar på en specifik position är satta (1). Anledningen till att 3 AND 2 ger 2 är att 3 och 2 har en gemensam etta på den position som är värd 2, men ingen annan stans.

Sanningstabellen för AND ser ut så här:

Sant	AND	Sant	=	Sant
Sant	AND	Falskt	=	Falskt
Falskt	AND	Sant	=	Falskt
Falskt	AND	Falskt	=	Falskt

Alltså, allt utom sant och sant ger falskt.

NOT

`NOT` är en logisk operator som tar en operand. Tillåten operand ska vara uttryck som utvärderas till heltal, flyttal eller till sant/falskt (booleskt värde, `0` eller inte `0`).

Resultatet av `NOT 1` och `NOT 0` betraktas som sant, medan `NOT -1` betraktas som falskt.

OR

`OR` är en logisk operator som tar två operander. Tillåtna operander är uttryck som utvärderas till heltal, flyttal eller till sant/falskt. `OR` används antingen i tester tillsammans med kommandot `IF` och ger då ett booleskt värde (`0` eller `-1`) som svar, eller i bit-operationer och ger då ett heltal som svar.

Följande exempel visar hur `OR` används i tester med `IF`. Exemplet skriver ut något `OK` skärmen, eftersom det räcker att en operand utvärderas som sann för att operatorn `OR` ska ge sant som svar. Endast när alla operander utvärderas som falska ger `OR` falskt som svar.

```
10 A=1
20 B=2
30 IF A>1 OR B>1
   THEN PRINT "OK"
```

Rad 10 sparar värdet 1 i variabeln A.
Rad 20 sparar värdet 2 i variabeln B.
Rad 30 skriver OK på skärmen om antingen A och/eller B har ett värde som är större än 1, vilket är fallet.

Detta exempel utför bit-operationen `3 OR 2` (alltså `00000011 OR 00000010`) vilket ger svaret `3` (`00000011`).

```
10 PRINT 3 OR 2
```

Rad 10 skriver ut resultatet av 3 OR 2 på skärmen.

När `OR` används för bit-operationer tittar operatorn på samtliga bitar i de båda talen, och ger `1` som svar om någon av talens bitar på en specifik position är satta (`1`). Anledningen till att `3 OR 2` ger `3` är att `3` och/eller `2` har en etta på positionen värd `1` och positionen värd `2`.

Sanningstabellen för `OR` ser ut så här:

Sant	OR	Sant	=	Sant
Sant	OR	Falskt	=	Sant
Falskt	OR	Sant	=	Sant
Falskt	OR	Falskt	=	Falskt

Alltså, allt utom falskt och falskt ger sant.

KAPITEL 6: ÖVRIGA NYCKELORD

Övriga nyckelord

GO

Nyckelordet `GO` används tillsammans med nyckelordet `TO` synonymt med kommandot `GOTO`. Däremot är det inte tillåtet att skriva `GO SUB` i stället för `GOSUB`. I version 7.0 av Commodore BASIC kan `GO 64` försätta en Commodore 128 i Commodore 64-läge[24].

TO

Nyckelordet `TO` används i två sammanhang. Nyckelordet `GOTO` används för att utföra hopp i programkoden. Detta kan även uttryckas som `GO TO`, vilket ger samma resultat. Exempel:

```
10 PRINT "HEJ"        Rad 10 skriver HEJ på skärmen.
20 GO TO 10           Rad 20 hoppar tillbaka till rad 10, som körs igen.
```

Programmet skriver `HEJ` på skärmen tills knappen **Run Stop** trycks ned.

Dessutom är `TO` en del av `FOR-NEXT`-iteration. Exempel:

```
10 FOR A=0 TO 4       Rad 10 påbörjar en iteration från 0 till 4.
20 PRINT "HEJ"        Rad 20 skriver HEJ på skärmen.
30 NEXT               Rad 30 stänger iterationen.
```

Programmet skriver `HEJ` på skärmen fem gånger.

[24] Commodore 128 är helt kompatibel med Commodore 64. Datorn har tre olika lägen: Commodore 128-läge som är standardläget med Commodore BASIC 7.0, CP/M-läget (som utnyttjar datorns andra processor av typen Z80) som nås genom en speciell boot-disk, och Commodore 64-läget som bl.a. kan nås genom kommandot GO 64.

KAPITEL 7: INBYGGDA KONSTANTER

Inbyggda konstanter

Commodore BASIC 2.0 second release har en enda inbyggd konstant, `π`.

π

`π` (PI) är en konstant som innehåller det ungefärliga värdet av förhållandet mellan en cirkels omkrets och diameter. I Commodore BASIC 2.0 second release är värdet av `π` satt till `3.14159265`.

Följande kod räknar ut omkretsen på en cirkel vars diameter är 5:

```
PRINT π*5
```

Svaret blir `15.7079633`. Och följande kod räknar ut omkretsen på en cirkel vars radie är 4:

```
PRINT (π*2)*4
```

Svaret blir `25.1327412`.

KAPITEL 8: SYSTEMVARIABLER

Systemvariabler

Till skillnad från en användardefinierad variabel får systemvariablerna sitt värde av systemet. Även om man inte använder systemvariablerna i sitt program är det viktigt att känna till att systemvariablerna finns. Commodore BASIC 2.0 second release har tre systemvariabler: `STATUS`, `TIME` och `TIME$`. Under kapitlet om datatyper påpekades att variabel identifieras av sina två första tecken, vilket innebär att du kan referera till `TIME` genom att skriva `TI` eller `TIMMY`. Följande kod ger felet *syntax*:

```
STEVE=10
```

Det beror helt enkelt på att variabeln `STEVE` identifieras genom sina två första tecken, `ST`, vilket även är sant för systemvariabeln `STATUS`. `STATUS` får sitt värde från systemet och inte från programmeraren, så att tilldela ett värde till `STEVE` orsakar samma fel som att försöka tilldela ett värde till `STATUS` – den gemensamma nämnaren är `ST`.

STATUS

Variabeln `STATUS` (eller `ST`) innehåller information om den senaste I/O-operationen. Variabeln innehåller åtta statusflaggor, en per bit. Flaggornas betydelse beskrivs i kapitlet om Commodore BASIC 2.0 DOS. Om man vet vad respektive bit är värd (1, 2, 4, 8, 16, 32, 64 eller 128) kan man kontrollera huruvida talet innefattas i värdet av `STATUS`.

Om `STATUS` ger `0` är ingen flagga satt. Om `STATUS` ger `4` är den tredje flaggan satt. Om `STATUS` ger `17` är den första (`1`) och den femte (`16`) satt. Man kan kontrollera en individuell flagga med den logiska operatorn `AND`. Resultatet av `17 AND 1` är `1`, resultatet av `17 AND 16` är `16`. Men testar man `17` mot `2`, `4`, `8`, `32`, `64` och `128` ger `0`.

TIME

Den inbyggda variabeln `TIME` anger under hur många *jiffies* (sextionsdels sekund) datorn har varit i gång. Om `TIME` innehåller värdet `92611` så har datorn varit i gång i drygt 1543,5 sekunder (eller knappt 26 minuter).

TIME$

Systemvariabeln `TIME$` anger hur mycket systemklockan är, i strängformat enligt: `HHMMSS` (`HH` = timme, `MM` = minut, `SS` = sekund). Variabeln `TIME$` får alltså ett nytt värde en gång i sekunden av systemet, och datorn antas ha

startats kl. 12:00 på natten (`000000`). I likhet med andra systemvariabler (`STATUS` och `TIME`) får `TIME$` sitt värde från systemet, men till skillnad från andra systemvariabler kan `TIME$` sättas av programmeraren. På det viset kan din dator hålla reda på den exakta tiden efter att klockan har ställts.

Applikationer som behöver hålla koll på tiden (till exempel OLF – order, lager, fakturering) kan fråga användaren efter aktuell tid vid uppstart.

Figur 16: Ställ klockan med TIME$.

Eventuell felaktig inmatning orsakar felet *illegal quantity*.

KAPITEL 9: ANVÄNDARDEFINIERADE FUNKTIONER

Användardefinierade funktioner

I sin allra enklaste form kan man använda GOSUB för att hoppa till en angiven del av programmet, där något utförs innan programmet fortsätter. DEF används för att skapa matematiska funktioner som anropas utan hopp med kommandot FN. Om man skriver en rutin i maskinkod, kan den anropas med kommandot SYS.

GOSUB är främst tänkt att använda för rutiner som utför något och använder och/eller manipulerar programmets variabler.

DEF används främst för matematiska funktioner, och kan inte innehålla några programsatser, endast uttryck. Funktionerna tar ett numeriskt argument och ger ett numeriskt resultat.

Likt GOSUB tar SYS ett numeriskt argument, men i stället för ett radnummer tar SYS en minnesadress som pekar på en maskinkodsrutin.

Dessa fem nyckelord står till ditt förfogande när du jobbar med användardefinierade funktioner:

- GOSUB
- DEF
- FN
- USR
- SYS

GOSUB

Kommandot `GOSUB` hoppar till en angiven del av ett program (radnummer), med möjlighet att återgå till raden efter anropet. Eftersom det är radnumret som anger vart programkörningen ska hoppa, är det lämpligt att skriva sina funktioner på höga radnummer och anteckna vilket radnummer som avser vilken funktion. I detta exempel ligger funktionen som dubblar värdet av `X` på rad 1000, vilket betyder att `X` dubblad varje gång `GOSUB 1000` anropas. Kommandot `RETURN` markerar att funktionen exekverat klart.

```
10 DIM X              Rad 10 skapar flyttalsvariabeln X.
20 X=32               Rad 20 initierar X med 32.
30 GOSUB 1000         Rad 30 anropar en funktion som dubblar X till 64.
40 GOSUB 1000         Rad 40 anropar samma funktion som dubblar X till 128.
50 PRINT X            Rad 50 skriver ut X (128).
60 END                Rad 60 avslutar programmet.
1000 X=X*2            Rad 1000 dubblar värdet i X.
1010 RETURN           Rad 1010 deklarerar funktionens slut och återgår till raden efter
                      anropet.
```

Detta är exekveringsordningen:

```
1       10 DIM X
2       20 X=32
3       30 GOSUB 1000
6       40 GOSUB 1000
9       50 PRINT X
10      60 END
4, 7    1000 X=X*2
5, 8    1010 RETURN
```

Se även `GOTO`, som precis som `GOSUB` hoppar till en angiven del av ett program, men utan möjlighet att hoppa tillbaka med `RETURN`. Alltså, ett hopp med `GOSUB` registreras i datorns call stack, ett hopp med `GOTO` registreras inte där. Om fler än 23 hopp lagras i datorns call stack, uppstår felet *out of memory*. Följande program brukar användas för att illustrera en oändlig iteration:

```
10 PRINT "HEJ"        Rad 10 skriver HEJ på skärmen.
20 GOTO 10            Rad 20 hoppar tillbaka till rad 10.
```

Följande kod (nästa sida) fyller call stacken med anrop (`GOSUB`) som aldrig plockas bort (med `RETURN`). Således skriver programmet ut 1 till 24 på skärmen. Efter att 24 skrivits ut försöker programmet hoppa till rad 20, men eftersom call stacken är full, uppstår felet *out of memory*.

```
10 A=0
20 A=A+1
30 PRINT A
40 GOSUB 20
```

DEF

Nyckelordet DEF används tillsammans med FN för att definiera en funktion för användning i ett program. FN utan DEF används sedan för att anropa funktionen. Funktionen kan beskrivas av andra funktioner, konstanter och vissa operatorer.

En med DEF definierad funktion tar alltid ett numeriskt argument och ger alltid ett numeriskt svar. Tänk på att ett namn identifieras av sina två första tecken, så du kan inte ha en funktion som heter APA om du också har en variabel som heter APP.

När du skapar dina funktioner, tänk på att du ska formulera dem som uttryck som tar ett numeriskt argument och som ger ett numeriskt svar enligt följande:

```
[RADNUMMER] DEF FN [NAMN](X)=[FUNKTIONSKROPP]
```

Detta program skapar en funktion – DBL – som dubblar värdet av parametern, och som sedan anropar funktionen DBL på nästa rad:

```
10 DEF FN DBL(N)=N+N
20 PRINT FN DBL(30)
```

Programmet ger 60 som svar.

FN

FN används för att anropa en användardefinierad BASIC-funktion eller tillsammans med DEF för att definiera en BASIC-funktion. Detta exempel skapar en funktion som fyrdubblar ett värde, genom att bygga vidare på föregående exempel:

```
10 DEF FN DBL(N)=N+N
20 DEF FN QUD(N)=FN DBL(N)*2
30 PRINT FN QUD(10)
```

Svaret blir 40, eftersom 40 är fyra gånger så stort som tio.

USR

Denna bok ger bara en ytlig överblick över `USR`. Man kan se på `USR` som en variant på `FN` med skillnaden `USR` anropar en maskinkodsfunktion i stället för en BASIC-funktion. Och till skillnad från `FN`, som anger att man vill anropa en av flera funktioner som sedan tidigare definierats med `DEF FN`, representerar `USR` en enda maskinkodsfunktion. Om `USR` anropas utan att ha konfigurerats, ger den felet *illegal quantity*.

Hur en funktion definieras, beror på vilken dator du använder. En VIC-20 tittar efter adressen till funktionen som `USR` anropar på adress `1-2` och en Commodore 64 tittar på adress `785-786` (låg byte först). Man kan antingen peka ut en egen maskinkodsfunktion eller någon befintlig rutin.

En fördjupning presenteras i nummer 38 av tidningen *Compute!* från 1983. De har en listning för både VIC-20 och Commodore 64 som skapar en användardefinierad maskinkodsfunktion, och ett litet BASIC-program som kopplar maskinkodsfunktionen till `USR` och låter användaren anropa funktionen med `USR`.

Maskinkodsfunktionerna listas som `DATA`-satser, men assemblerkoden ser ut så här för Commodore 64 (från adress `828` eller `33C` i hexadecimal):

```
JSR $BC9B
LDA $64
STA $FC
LDA $65
STA $FB
LDY #$00
LDY ($FB),Y
TAY
LDA #$00
JSR $B391
RTS
```

Även en version för VIC-20 är tillgänglig i tidningen. Så vad gör maskinkodsfunktionen som listas ovan? Den tar helt enkelt en minnesadress och ger byten som ligger lagrad på den adressen som svar. Exempelprogrammet som använder funktionen börjar med att peka ut `828`[25] som `USR`-adress. Förutsatt att maskinkodsfunktionen är på plats görs detta så här på VIC-20:

```
POKE 1,60
POKE 2,3
```

Igen, förutsatt att maskinkodsfunktionen är på plats görs detta så här på Commodore 64:

```
POKE 785,60
POKE 786,3
```

Funktionen kan därefter anropas så här:

```
PRINT USR(828)
```

Anropet ger svaret `32`, vilket är operationskoden för `JSR` som lagras på adress `828`. Men vilken minnesadress som helst kan anges, och `USR` svarar med det värde som ligger på adressen som anges, alltså precis som `PEEK` gör.

Här är en fungerande programlistning för Commodore 64 som matar in maskinkodsfunktionen, pekar ut den och anropar den med `USR` (nästa sida):

[25] Varför `60` och `3` är `828` beskrivs i kapitlet om 16-bitarstal.

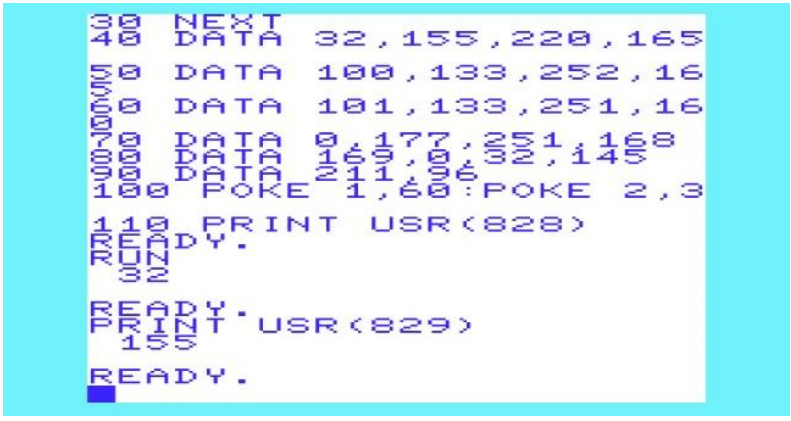
```
10 FOR A=828 TO 849
20 READ D:POKE A,D
30 NEXT
40 DATA 32,155,188,165
50 DATA 100,133,252,165
60 DATA 101,133,251,160
70 DATA 0,177,251,168
80 DATA 169,0,32,145
90 DATA 179,96
100 POKE 785,60:POKE 786,3
110 PRINT USR(828)
```

Rad 10 itererar genom DATA-satserna som innehåller maskinkodsprogrammet.
Rad 20 läser in DATA-satserna och lagrar dem i minnet.
Rad 30 stänger iterationen som inleds på rad 10.
Rad 40-90 innehåller maskinkodsprogrammet som listas på föregående sida.
Rad 100 berättar för USR var i minnet maskinkodsfunktionen finns.
Rad 110 anropar maskinkodsfunktionen, som berättar vad som finns lagrat på minnesadress 828.

Efter att programmet har körts, kan `USR` anropas fler gånger med andra adresser än 828, efter tycke och smak. Här är en fungerande programlistning för VIC-20 som gör samma sak:

```
10 FOR A=828 TO 849
20 READ D:POKE A,D
30 NEXT
40 DATA 32,155,220,165
50 DATA 100,133,252,165
60 DATA 101,133,251,160
70 DATA 0,177,251,168
80 DATA 169,0,32,145
90 DATA 211,96
100 POKE 1,60:POKE 2,3
110 PRINT USR(828)
```

Rad 10 itererar genom DATA-satserna som innehåller maskinkodsprogrammet.
Rad 20 läser in DATA-satserna och lagrar dem i minnet.
Rad 30 stänger iterationen som inleds på rad 10.
Rad 40-90 innehåller maskinkodsprogrammet som listas på föregående sida.
Rad 100 berättar för USR var i minnet maskinkodsfunktionen finns.
Rad 110 anropar maskinkodsfunktionen, som berättar vad som finns lagrat på minnesadress 828.



Figur 17: USR på VIC-20

SYS

Kommandot `SYS` används för att från BASIC starta ett program eller en rutin som är skriven i maskinkod. `SYS` tar ett argument som är minnesadressen där programmet eller rutinen finns lagrad i minnet. Om adressen undviks uppstår felet *syntax*, och om argumentet är av fel typ uppstår felet *type mismatch*. Om argumentet är av rätt typ (heltal) men ligger utanför tillåtet omfång, uppstår felet *illegal quantity*. Maskinkodsprogram avslutas med instruktionen RTS. `SYS` beskrivs i kapitlet om användardefinierade funktioner, men det finns en del färdiga rutiner i din dator som kan nås genom att anropa SYS.

Vill du starta om din Commodore 64 (*soft reset*), skriv:

```
SYS 64738
```

I Commodore BASIC 7.0 tar SYS flera argument för att initiera datorns register, men på VIC-20 och Commodore 64 görs detta med POKE vid behov, vilket beskrivs i din dators manual.

Se även *soft reset* och *hard reset* i ordförklaringarna i appendix C.

KAPITEL 10: 16-BITARSTAL

16-bitarstal

Commodore BASIC 2.0 second release kan utan problem hantera väldigt stora tal. Men ibland måste man skriva tal direkt till datorns RAM-minne. Varje minnesadress i datorns RAM håller en byte (alltså åtta bitar), vilket innebär att varje byte kan hålla en av 256 olika konfigurationer av satta eller icke satta bitar. Om man tolkar dessa olika konfigurationer som tal mellan 0 och 255 så är det ganska enkelt att skriva och läsa data. Man använder kommandot `POKE` för att skriva det önskade talet mellan 0 och 255 på valfri adress och man använder funktionen `PEEK` för att läsa av vilket tal som ligger lagrat på en specifik minnesadress. Det binära talsystemet fungerar som det decimala talsystemet, men det binära talsystemet har 2 som bas i stället för 10. Så här ser de tio första talen ut i binärform:

Värde i decimalform	Värde i binärform
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

(För det mesta presenteras ett åttabitarstal i binärform med inledande nollor, så att man ser att det är ett åttabitarstal. 9 i binärform skrivs till exempel `00001001` eller `0000 1001` så att man ser den höga och låga nibbeln tydligt.) Och när vi kommer upp till det 256:e talet, alltså 255, är alla tillgängliga bitar satta.

Värde i decimalform	Värde i binärform
246	11110110
247	11110111
248	11111000
249	11111001
250	11111010
251	11111011
252	11111100
253	11111101
254	11111110
255	11111111

Vill jag skriva värdet `100` på adress `828` skriver jag:

```
POKE 828,100
```

Och för att läsa av värdet jag nyss skrev, kan jag skriva:

```
PRINT PEEK(828)
```

...vilket ger `100` som svar.

Värdet 100 kan representeras som 8-bitarstal, vilket betyder att värdet kan beskrivas med endast en byte. Men ibland behöver man skriva in ett 16-bitarstal. Genom att kombinera två bytes på vardera åtta bitar får man 16 bitar. Antalet unika kombinationer av satta eller icke satta bitar blir då 65536 stycken, vilket innebär att man kan beskriva tal mellan 0 och 65535.

Precis som 6 i talet 637 är värt mer än 7 i 637, så har man valt att placera den byte som har lägst värde först när man parar ihop två bytes. Så om vi förväntas skriva ett 16-bitarstal på adress 828, så skriver vi både till adress 828 och adress 829. Din Commodore 64 eller VIC-20 kommer att tolka talet genom att läsa den höga byten, multiplicera den med 256 och addera den låga byten.

16-bitarsversionen av talet 100 (som alltså går att beskriva som 8-bitarstal) blir alltså `100` och `0` eftersom 0×256 + 100 = 100. Talet `257` lagras som `1` och `1` eftersom 1×256 + 1 = 257.

Detta program lagrar 16-bitarstalet `828` på adress `875`:

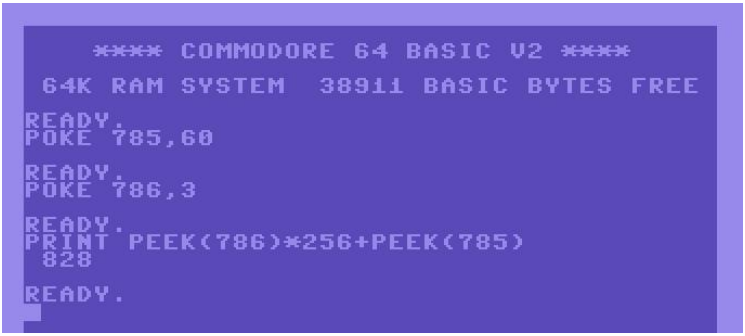
```
POKE 785, 60
POKE 786, 3
```

Anledningen till att `828` består av `60` och `3` är att 3×256 är 768 och 768 + 60 är 828. Du kan läsa av 16-bitarstalet genom att skriva:

```
PRINT PEEK(786)*256+PEEK(785)
```

Och i teorin kan man skapa 24-bitarstal eller 32-bitarstal på samma vis, men för dig som programmerar BASIC finns det ingen anledning eftersom Commodore BASIC 2.0 second release kan hantera mycket stora flyttal. Du kan dubbla siffran 1 hela 125 gånger utan att stöta på problem (men inte en 126:e gång).

När 1 har dubblerats en gång har man värdet 2, och när 1 har dubblerats 4 gånger har man värdet 16.



Figur 18: Skrivning och avläsning av ett 16-bitarstal.

När 1 har dubblerats 20 gånger har man värdet 1 048 576, och när 1 har dubblerats 125 gånger kommer din dator att visa `4.25352959E+37`. Det motsvarar exakt följande tal:

42 535 295 865 117 307 932 921 825 928 971 026 432

Den visuella representationen innehåller inte alla detaljer, för det finns en begränsning i vilken precision som klaras av. En översikt får man genom att titta på siffran före punkten, `4`, och siffran efter `E+`, `37`. Det säger att talet motsvarar ungefär en fyra och 37 nollor, alltså ungefär 40 miljoner kvintiljoner. Siffrorna efter punkten och före `E+` ger de detaljer som datorn har. Alltså, siffran efter 4 är 2 och därefter kommer 5, och så vidare.

Om du dubblerar 1 trettio gånger visar datorn `1.07374182E+09`, vilket är datorns representation av talet 1 073 741 824 (drygt en miljard). Detta är för att en miljard är en etta (före punkten) med nio nollor (efter `E+`). Siffrorna där emellan är den precision som datorn klarar av att visa.

Varje positiva 16-bitarstal som är mindre än 256 representeras av sig själv följt av 0. För att räkna ut vilka två 8-bitarstal som representerar ett givet 16-bitarstal som är större än 255, kallat `X` nedan, använd följande metod:

För att få fram första byten (den minst viktiga): `X` mod 256. Notera resultatet.

För att få fram den andra byten (den mest viktiga): `X/256`. Notera heltalsdelen av resultatet.

I fallet 828 ger det oss följande resultat:

828 mod 256 = 60 (eller 00111100 i binär form).

828 / 256 = 3 (eller 00000011 i binär form).

Den binära representationen av 828 är (uppdelat på nibbles) är...

```
0000 0011 0011 1100
```

...för att den binära representationen av 60 är (uppdelat på nibbles) är...

```
0011 1100
```

...och den binära representationen av 3 är (uppdelat på nibbles) är...

```
0000 0011
```

Eftersom Commodore-maskiner lagrar den minst viktiga byten först, kommer 828 att representeras så här i datorns arbetsminne (uppdelat på bytes):

```
00111100 00000011
```

För ett konkret exempel, se appendix F om maskinkod.

Big endian eller little endian?

Svaret på frågan om *big endian* eller *little endian* handlar om vilken ordning bytes ska placeras i de fall talet i fråga består av fler än en byte. Termologin är lånad från boken Gullivers resor och handlade då om vilken ända av ett ägg som ska vara överst i en äggkopp - den smala (little endian) eller den breda (big endian)? Översätter man detta till datorvärlden så handlar det om var den mest signifikanta byten i ett tal som är större än ett 8-bitarstal ska placeras.

Tänker man på hur 16-bitarstal lagras i minnet på en Commodore-maskin, så kan man konstatera att den minst viktiga byten kommer först.

Betrakta exemplet ovan. Vi kan konstatera att talet 828 beskrivs med två bytes. Dessa är 60 (00111100) och 3 (00000011). Vi kan konstatera att den första byten bidrar med sitt värde, medan den andra är mer signifikativ, eftersom den bidrar med sitt värde multiplicerat med 256. Hade vi pratat om

ett 32-bitarstal så hade byte nummer tre bidragit med sitt värde multiplicerat med 65536 och byte nummer fyra med sitt värde multiplicerat med 16777216. Den första byten är minst signifikativ eftersom den är värd minst - den är värd sitt eget värde, varken mer eller mindre. Den lilla änden först alltså, och således little endian. Ägget är en god hjälp att komma ihåg detta. Little endians ville ha äggets smalaste sida upp, så att den minsta delen nås först, vilket är fallet i din Commodore-maskin: Den minst signifikanta byten först.

Så åter igen, för att komma fram till 828 med byten 60 följt av byten 3, är det 60 som ska adderas till resultatet som det är, men det är 3 som måste multipliceras med 256 för att nå sin rätta signifikans - lilla änden först. 60 är värt 60, 3 är värt 768 (alltså 3×2) och summan av 60 och 768 är 828.

Figur 19: Little endian. Foto: Marie-Lan Nguyen

Detta säger oss att det största heltalet på 32 bitar (fyra bytes) är 4 294 967 295. Det högsta värdet för varje byte är 255. Den högsta värderepresentationen för varje byte är 256, 256×256, 256×256×256 och 256×256×256×256. Om man tar i beaktning att 0 tar upp en plats i den tillgängliga kombinationen, måste 1 alltid dras av från varje summa. Alltså (256+65 536-1) för sextonbitstal, (16 777 216-1) för tjugofyrabitarsstal och (4 294 967 296-1) för trettiotvåbitarstal. Vi kan alltså med ganska lite förbrukat minne beskriva väldigt stora heltal. Ursäkta mitt språköras inkonsekvens.

KAPITEL 11: METODER

Metoder

Detta kapitel visar lösningar på två vanliga programmeringsutmaningar i Commodore BASIC 2.0 second release. Kapitlet tar upp följande:

- Bakgrundsfärg och borderfärg
- Hur man skapar en meny för programmets användare
- Prestandamätning
- Spara minne på disk
- Bitmaskning

Bakgrundsfärg och borderfärg

Både Commodore 64 och VIC-20 kan visa 16 färger. Dessa är svart, vit, röd, turkos, lila, grön, blå, gul, orange, brun, ljusröd, mörkgrå, grå, ljusgrön, ljusblå och ljusgrå. Bakgrundens färg kontrolleras på Commodore 64 av adress `53281` (`D021` i hexadecimal), där `0` betyder svar, `1` betyder vit, och så vidare. Följande exempel fungerar endast på Commodore 64. Detta ger svart bakgrundsfärg:

```
POKE 53281,0
```

Det är endast de fyra låga bitarna (som beskriver värdet 0-15) som används, vilket innebär att samma resultat uppnås om talet 16 skrivs till adress `53281`, då 16 och 0 har samma fyra låga bitar – `0000` – nämligen `00010000` respektive `00000000`.

Borderfärgen (ramfärgen) har samma färgpalett som bakgrundsfärgen. Borderfärgen kontrolleras av adress `53280` (`D020` i hexadecimal), och även här är det bara låg nibble som används. Detta ger gul ram:

```
POKE 53280,7
```

Commodore 64:s färgpalett är:

Färgkod	**Motsvarande färg**
0	Svart
1	Vit
2	Röd
3	Turkos
4	Lila
5	Grön
6	Blå
7	Gul
8	Orange
9	Brun
10	Ljusröd
11	Mörkgrå
12	Grå
13	Ljusgrön
14	Ljusblå
15	Ljusgrå

VIC-20 använder adress `36879` (`900F`) för både borderfärg (bit 0 till 2), bakgrundsfärg (bit 4-7) och inverterat läge (bit 3). Inverterat läge innebär att

tecken visas med bakgrundsfärgen och att varje tecken ramas in av teckenfärgen.

Då endast tre bitar (0 till 2) används till bordern, är det bara en av de första åtta färgerna (svart, vit, röd, turkos, lila, grön, blå eller gul) som kan användas där. Följande exempel fungerar endast på VIC-20. Detta ger blå border och gul bakgrund:

```
POKE 36879,126
```

Den binära representationen av 126 är `01111110`. Bit 0-2 (`110`) beskriver värdet 6, vilket betyder blå borderfärg. Bit 3 har värdet 1 vilket betyder normalt (ej inverterat) läge. Och bit 4-7 (`0111`) beskriver värdet 7, vilket betyder gul bakgrundsfärg. Detta väljer samma färger i inverterat läge:

```
POKE 36879,118
```

Den binära representationen av 118 är `01110110`. Bit 0-2 (`110`) beskriver värdet 6 (blå border). Bit 3 har värdet 0 vilket betyder inverterat läge. Och bit 4-7 (`0111`) beskriver värdet 7 vilket betyder gul bakgrund, men i inverterat läge är det i stället *texten* som är gul, medan teckenfärgen ramar in varje tecken.

VIC-20:s färgpalett är:

Färgkod	**Motsvarande färg**
0	Svart
1	Vit
2	Röd
3	Turkos
4	Lila
5	Grön
6	Blå
7	Gul
8	Orange
9	Ljusorange
10	Rosa
11	Ljusturkos
12	Ljuslila
13	Ljusgrön
14	Ljusblå
15	Ljusgul

Låt oss strunta i VIC-20:s inverterade läge och formulera en metod att sätta

både bakgrundsfärg och borderfärg. Givet att bakgrunden bestäms av bitarna 4, 5, 6 och 7 så måste färgkoden för bakgrundens färgkod förstoras 16 gånger, och givet att vi vill att bit 3 (som viktas till 8) alltid ska vara satt, sätter följande program önskad bakgrundsfärg (0-15) och borderfärg (0-7) till ljusorange (9) respektive gul (7):

```
10 BAKGR=9
20 BO=7
30 POKE 36879,BAKGR*16+BO+8
```

Rad 10 väljer bakgrundsfärg 9 (ljusorange).
Rad 20 väljer borderfärg 7 (gul).
Rad 30 skriver 9 till adress 36879 bit 4-7 och 7 till samma adress

Notera att variabeln som håller bakgrundsfärgen inte kan heta `BAKGRUND` eftersom `RUN` (som är en del av ordet) är ett reserverat nyckelord. Av samma skäl kan inte variabeln som håller borderfärgen heta `BORDER` då `OR` är ett reserverat nyckelord. Tänk också på att variabler identifieras av sina två första tecken, så rad 10 sätter *alla* variabler som börjar på `BA` till `9`.

Resultatet av körningen borde ge en gul borderfärg och en ljusorange bakgrundsfärg. Se delen om bitmaskning senare i detta kapitel för mer information.

Meny

Kommandot `GET` sparar värdet av den tangent som just nu trycks ner i den variabel som anges som argument. `GET A$` lagrar ett blanksteg i `A$` givet att kommandot körs exakt när blankstegstangenten är nedtryckt. Så för det mesta kommer `A$` att vara tom. Den kortaste koden för att invänta en tangenttryckning och lagra dess värde i `A$` får man om man placerar `IF`-satsen på samma rad som `GET`.

```
10 GET A$:IF A$="" GOTO 10
```

Jagar man bytes på allvar, så går det att ta bort alla blanksteg. Ett enkelt program som låter användaren antingen växla färger på skärmen eller avsluta, skulle alltså kunna se ut så här (endast VIC-20):

```
10 RESET=PEEK(36879)
20 PRINT "1. ÄNDRA FÄRGER"
30 PRINT "2. AVSLUTA"
40 GET A$:IF A$="" GOTO 40
50 IF A$="1" THEN POKE 36879,PEEK(36879)+1
60 IF A$="2" THEN POKE 36879,RESET:END
70 GOTO 40
```

Efter att ha läst avsnittet om färger i detta kapitel kan du säkert anpassa programmet så att det fungerar på Commodore 64 istället.

> Apropå `IF`-satsen på rad 40: Kom ihåg att `THEN GOTO 40` kan kortas ner till antingen `THEN 40` eller `GOTO 40`. Det som avses i båda fallen är `THEN GOTO 40`.

Prestandamätning

Utrustad med kunskap om systemvariabeln `TIME` kan du mäta hur lång tid en process tar att exekvera, enligt följande:

1. Spara värdet i `TIME` i en variabel.

2. Utför det du vill mäta.

3. Skriv ut differensen mellan `TIME` och variabeln. Detta ger dig antalet förbrukade jiffys.

Följande kod visar hur lång tid det tar för din dator att skriva ut `HEJ` på skärmen 100 gånger[26]:

```
10 S=TI
20 FOR A=1 TO 100
30 PRINT "HEJ"
40 NEXT
50 PRINT TI-S
```

Rad 10 sparar antal jiffys som datorn har varit igång i variabeln S[27].
Rad 20 öppnar en iteration som ska köras 100 gånger.
Rad 30 skriver ut HEJ, vilket kommer att ske 100 gånger.
Rad 40 stänger iterationen.
Rad 50 skriver ut hur många jiffys som passerat under den tid programmet har exekverat.

Programmet skriver ut `HEJ` på skärmen 100 gånger, och avslutar med att skriva ut antal jiffys det gick åt att skriva `HEJ` på skärmen 100 gånger.

På VIC-20 får du ett värde mellan `93` och `113`, beroende på hur många gånger datorn behöver rulla textskärmen för att utföra uppgiften. På Commodore 64 får du ett värde mellan `180` och `228`.

Är du nyfiken på antalet sekunder värdet motsvarar, behöver du bara dividera det med 60.

[26] `https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/speedtest.bas`

[27] Notera att `TI` är samma sak som `TIME` eftersom Commodore BASIC 2.0 second release endast tittar på en variabels två första tecken i namnet.

Spara minne på disk

Som nämndes i avsnittet om `SAVE` i kapitlet om kommandon, så finns det inget bra sätt att spara en del av minnet på kassett i Commodore BASIC 2.0 second release, men det finns ett par strategier man kan vidta.

Oavsett vilken strategi som väljs är det viktigt att komma ihåg att Commodore DOS (se kapitel 16) har stöd för olika filtyper. Om ingen filtyp anges, så antas `PRG` gälla (program-fil). När man handskas med filer på låg nivå så spelar detta ingen roll, men din diskdrive kommer att göra vissa antaganden om filen, beroende på dess typ.

Framför allt antas programfiler avsätta två bytes först i filen för att deklarera var informationen ska läsas in - en programfil fungerar inte som den ska om den inte läses in till samma minnesadress den sparades ifrån, medan t.ex. en fil som innehåller t.ex. en sprite kan tänkas läsas in till vilket lämpligt minnesområde som helst.

Avsnittet tar upp två strategier för att korrekt spara minne på kassett eller diskett.

- Strategi 1, att skriva data direkt till en fil, är det mest komplicerade sättet, men det fungerar oavsett om du arbetar med diskett eller kassett, och oavsett vilken typ av data du skriver ner.
- Strategi 2, att använda det sekundära numret, är enklare men fungerar bara för programfiler och endast på diskett.

Strategi 1: Skriv data direkt till en fil

Minns att `OPEN` accepterar information om filtyp (beskrivs i kapitlet om DOS) och läge (läs eller skriv) i samma argument som filnamnet. Om det är maskinkod du skriver (typ `P`) ska de två bytes i filens inledning berätta var i minnet informationen är hämtat ifrån, och således vart informationen ska landa vid tillbakaläsning. Annars (rimligtvis typ `S`) kan filen innehålla sitt data, varken mer eller mindre.

Följande program fungerar endast på Commodore 64, och förutsätter att du har en diskett med lite ledig plats i drive `8`, och att disketten inte innehåller någon fil som heter `PROGRAM`. Betrakta programmet på nästa sida.

```
10 ADR=8192
20 LNGT=1
30 POKE ADR,96
40 OPEN 1,8,1,"PROGRAM, SEQ, W"
50 PRINT#1,CHR$(ADR AND 255);
60 PRINT#1,CHR$(ADR/256);
70 FOR A=0 TO LNGT-1
80 PRINT#1,CHR$(PEEK(ADR+A));
90 NEXT
100 CLOSE 1
```

Programmet skriver ett maskinkodsprogram på disk, och den första raden, rad `10`, anger adressen som programmet ska sparas på. För att anpassa programmet till VIC-20, ändra `8192` till `828`.

Rad `20` anger längden på programmet, vilket är 1 byte. Den enda instruktionen som utgör vårt maskinkodsprogram är 96, som direkt avslutar programmet.

Rad `30` skriver instruktionen till minnet.

Rad `40` öppnar en sekventiell fil för skrivning - se `OPEN` för mer information.

Rad `50-60` skriver minnesadressen som programmet ska återkallas till när det läses in med `LOAD`, den minst signifikativa byten först - se kapitlet om 16-bitarstal för mer information. Dessa två rader ska inte vara med om du skriver annan data, t.ex. grafik, till filen.

Rad `70-90` kopierar innehållet i minnet, från startadress fram till önskad längd, till filen, som sedan stängs på rad `100`.

Kör BASIC-programmet för att skapa filen med maskinkodsprogrammet på disketten. Rent tekniskt innehåller filen nu tre bytes: De två första är programmets startadress och den tredje är själva programmet med instruktionen att avsluta sig själv.

Exemplet på nästa sida förutsätter att du har en Commodore 64, men ändra `8192` till `828` om du har en VIC-20.

Givet att du har en Commodore 64, stäng av din dator, eller nollställ minnesadress `8192` genom att skriva `POKE 8192,0`. När du nu testar adress `8192`, får du `0` som svar.

```
PRINT PEEK(8192)
 0

READY.
```

Ladda sedan in programmet i minnet med `,8,1`, och konstatera att det lagt sig på adress `8192`.

```
LOAD"PROGRAM",8,1

SEARCHING FOR PROGRAM
LOADING
READY.
PRINT PEEK(8192)
 96

READY.
```

Du kan rent av starta programmet, och konstatera att det direkt avslutas.

```
SYS 8192

READY.
```

Nästa sida visar den andra strategin för att spara minne på diskett: Att göra det som `SAVE` är tänkt att göra, men inte gör.

Tänk på att:

Programfiler som skapas på detta vis, måste ha en angiven startadress skriven först i filen, vilket rad 50-60 sköter om. För övriga datafiler, utelämna startadressen.

Om det är specifikt programfiler som ska skrivas, och om dessa specifikt ska skrivas på disk, kan den enklare metoden (nästa sida) användas istället.

Strategi 2: Använd det sekundära numret

Denna strategi är endast tillgänglig för den som sparar på diskett när adressen för återkallande ska lagras först i filen, vilket typiskt är relevant för maskinkodsprogram. För andra scenarier, se strategi 1.

När kommandot `LOAD` används med tillsammans med det sekundära numret `1` så antas de två bytes som kommer först, inte vara en del av det data som ska läsas in, utan information om var i minnet resten av filens data ska läsas in. Detta är typiskt för programfiler.

När man sparar med `SAVE` så anger det sekundära numret `1` att vi vill:

- Spara en specificerad del av minnet som ett program
- Att startadressen ska skrivas först i filen

Av olika skäl kommer vi inte använda nyckelordet `SAVE`, utan anropa den underliggande rutinen med `SYS`. Precis som sist fungerar följande program endast på Commodore 64, och förutsätter att du har en diskett med lite ledig plats i drive `8`, och att disketten inte innehåller någon fil som heter `PROGRAM` - ersätt filnamnet på rad `40` om du behöver.

```
10 POKE 8192,96
20 FIRST=8192
30 LAST=8193
40 SYS 57812("PROGRAM"),8,1
50 POKE 193,FIRST AND 255
60 POKE 194,FIRST/256
70 POKE 174,LAST AND 255
80 POKE 175,LAST/256
90 SYS 62957
```

Rad 10 skriver det data vi vill spara till minnet.
Rad 20-30 beskriver det minnesomfång vi vill spara.
Rad 40 förbereder SAVE-rutinen med filnamn, enhetsnummer och sekundärt nummer.
Rad 50-80 skriver minnesomfånget till registret som SAVE-rutinen läser.
Rad 90 exekverar SAVE-rutinen.

Apropå rad `10`, tänk att `96` som lagras på `8192`, är ett minimalt maskinkodsprogram som innehåller en enda instruktion, nämligen att avsluta programmet. Sen använder vi `SYS` för att både konfigurera `SAVE`-rutinen och exekvera den. Din diskett ska nu innehålla en fil på totalt tre bytes, där de två första är 0 och 32, och den tredje är 96. Av följande anledning:

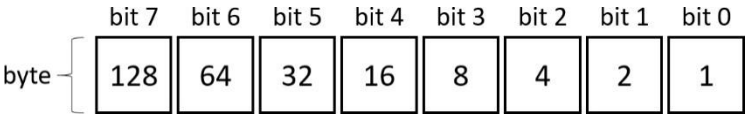
32×256 är 8192 och 8192 + 0 är fortfarande 8192, och det var adressen vi ville att programmet skulle återkallas till. Och programmet var just endast 96, vilket i detta sammanhang betyder "avsluta"

Bitmaskning

Med bitmaskning avses möjligheten att läsa av eller skriva till en individuell bit åt gången, trots att varje minnesadress består av åtta bitar. Användningsområdet bakom att kontrollera individuella eller några bitar åt gången i en byte skulle kunna vara:

- Du vill slå av eller på individuella sprites på Commodore 64
- Du vill hantera två fyrabitarstal i en enda byte (som tidigare nämndes, bakgrundsfärg och borderfärg på VIC-20 beskrivs av ett 3- och ett 4-bitarstal i en och samma byte)

Varje bit (från vänster till höger) representerar 128, 64, 32, 16, 8, 4, 2, och 1. De är numrerade från höger till vänster, vilket innebär att det är bit 0 som är värd 1, bit 1 som är värd 2, bit 3 som är värd 4, och så vidare.



Figur 20: Bitarna i en byte, med dess nummer och värde.

Läs en bit

För att läsa en bit måste du veta vad den representerar som satt. Om du vill läsa av bit nummer 4, är det dess värde (16) som du ska använda som mask. Detta kontrollerar om bit 4 är satt på adress `8192`. Om så, blir svaret bitens värde (16), annars 0.

```
PRINT PEEK(8192) AND 16
```

Sätt en bit till 1

För att sätta en bit till 1, skicka det befintliga värdet på adressen, modifierat med OR och bitens värde tillbaka till samma adress. För att sätta bit 6 (värd 64) på adress `8192` till 1, skriv:

```
POKE 8192,PEEK(8192) OR 64
```

Sätt en bit till 0

För att sätta en bit till 0 på en specifik position, utan att påverka andra bitar, behöver du ett tal vars bitmönster består av ettor förutom på den position där du vill sätta biten till 0. För att sätta bit 7 (längst till vänster, högst värde) till

0 behöver du använda dig av bitmönstret `01111111`. Det motsvarar talet 127, och koden blir då:

```
POKE 8192,PEEK(8192) AND 127
```

KAPITEL 12: TEXT

Text

Detta kapitel beskriver hur text hanteras på skärmen. Kapitlet berör följande kommandon och aspekter:

- Skiftläge
- Kommandot `PRINT`
- Val av enhet att skriva till
- Begränsningar
- Funktioner för textmanipulation
- Hämta input från användaren
- Omvandla mellan text och tal
- Textminnet

Skiftläge

Både VIC-20 och Commodore 64 har två teckenuppsättningar. Dessa är:

- Versaler och pseudografik
- Gemener och versaler

Datorn startar i läget med versaler och pseudografik. Pseudografiken är tecken som innehåller olika små bilder som man kan få fram genom att hålla ner antingen **Commodore** eller **Shift**. De tecken som skrivs när man håller nere **Commodore** finns tillgänglig även när man växlar till läget med gemener och versaler. För att växla vilken teckenuppsättning som används, tryck **Commodore+Shift**.

För att programmeringsmässigt växla läge manipulerar man pekaren till teckenminnet som ligger lagrad på adress `53272` på Commodore 64. För att växla till versaler och pseudografik (som är ursprungsläget), skriv `21` till adressen i fråga.

`POKE 53272,21` (endast Commodore 64)

Minnesadressen `53272` kommer du även i kontakt med i kapitlet om grafik. För att växla till gemener och versaler, skicka 23 till samma adress.

`POKE 53272,23` (endast Commodore 64)

På VIC-20 ligger pekaren lagrad på adress `36869`. För att växla till versaler och pseudografik (som är ursprungsläget), skriv `240` till `36869`.

`POKE 36869,240` (endast VIC-20)

För att växla till gemener och versaler, skicka `242` till samma adress.

`POKE 36869,242` (endast VIC-20)

Om ditt program är beroende av att en viss teckenuppsättning är vald, kan det vara relevant att stänga av användarens möjlighet att växla teckenuppsättning. För att stänga av möjligheten att växla med **Commodore+Shift**:

```
PRINT CHR$(8)
```

Och för att återställa, så att användaren själv kan växla igen:

```
PRINT CHR$(9)
```

Apropå pseudografik så är världens kortaste labyrintprogram faktiskt skrivet i Commodore BASIC 2.0 second release. Källkoden ser ut så här:

```
10 PRINT CHR$(205.5+RND(1));:GOTO 10
```

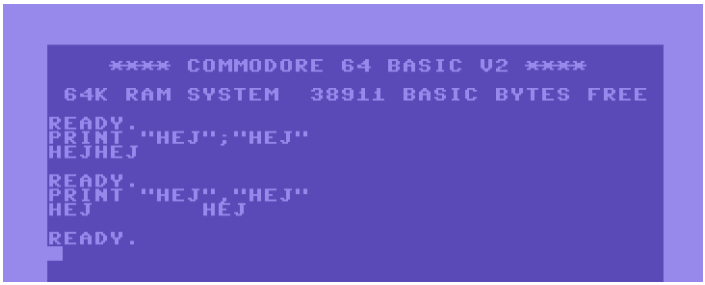
Så här ser en körning på VIC-20 ut:

Figur 21: Världens kortaste labyrintprogram.

Givet att din Commodore-dator befinner sig i läget för versaler och pseudografik är PETSCII-koden för en diagonal från vänster till höger `205` och för den spegelvända diagonalen `206`. Funktionen `CHR$` tar en PETSCII-kod och ger tecknet som representeras i retur (till `PRINT` i detta fall). Om ett flyttal skickas in, läses bara heltalsdelen, så `205.5` och `205.99` kommer båda att tolkas som `205`. `RND(1)` ger ett flyttal mellan `0` och (precis under) `1` som svar. Det innebär att summan av `205.5` och `RND(1)` kommer att tolkas som `205` i snitt hälften av gångerna och som `206` i snitt hälften av gångerna. Ett avslutande semikolon säger att nästa tecken ska skrivas på samma rad som föregående (såvida inte terminalen kräver ett radbryte). Således fyller programmet skärmen med snedstreck som tillsammans utgör en labyrint.

Kommandot PRINT

`PRINT` skriver ut valfritt uttryck, till exempel en konstant, värdet av en variabel eller resultatet av en beräkning eller returen av en funktion. Flera uttryck kan skrivas ut efter varandra om de avgränsas med semikolon (`;`) eller komma (`,`). Semikolon skriver ut resultatet av nästa uttryck direkt efter föregående medan komma skriver ut resultatet av nästa uttryck på nästa lediga tabulator-position, som är var tionde kolumn.



Figur 22: Avgränsning med semikolon och komma.

Blanksteg distribueras enligt följande:

- Före varje numerisk variabel skrivs ett blanksteg.
- Vid semikolon infogas ett blanksteg om värdet till vänster eller värdet till höger är numeriskt.

Det innebär att `PRINT 1;1` ger:

```
 1  1
```

`PRINT "A";1` ger:

```
A 1
```

`PRINT 1;"A"` ger:

```
 1 A
```

`PRINT "A";"A"` ger:

```
AA
```

Negativa tal inleds inte med blanksteg utan med ett minus-tecken, men `PRINT` *ser alltid till att det är minst ett blanksteg mellan ett tal och ett annat värde eller mellan två tal.*

Där uttryck kan skiljas åt utan avgränsare, behöver avgränsare inte användas – semikolon antas vara det som åsyftas. Det innebär att `PRINT "A" "B"` *ger samma resultat som* `PRINT "A";"B"`*.*

Val av enhet att skriva till

Kommandot `CMD` används för att styra om `PRINT` från skärmen till en annan enhet, till exempel en diskdrive eller en printer. Syntax:

```
CMD logiskt filnummer [, uttryck [, uttryck…]]
```

`CMD` kan användas antingen direkt eller i ett program, och det enhetsnummer som anges öppnas därefter med `OPEN`.

`CMD` beskrivs närmare i kapitlet om kommandon.

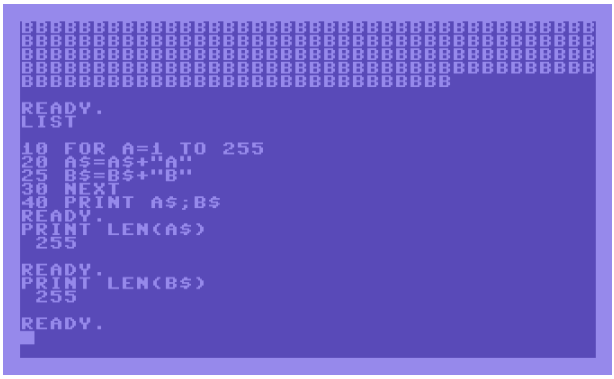
Begränsningar

En enskild textsträng kan innehålla upp till 255 tecken, vilket motsvarar knappt 12 rader på skärmen på VIC-20 och drygt 6 rader på Commodore 64 - funktionen `LEN` svarar på hur många tecken en sträng innehåller. Kommandot `PRINT` kan däremot skriva ut betydligt längre textsträngar än så, eftersom flera strängar kan skrivas ut efter varandra, avgränsat med semikolon. Följande program skapar två strängar innehållande 255 tecken vardera, och skriver ut dem tillsammans:

```
10 FOR A=1 TO 255
20 A$=A$+"A"
30 B$=B$+"B"
40 NEXT
50 PRINT A$;B$
```

Rad 10 räknar från 1 till 255, vilket innebär att rad 20 och 30 körs 255 gånger.
Rad 20 lägger till ett tecken i A$.
Rad 30 lägger till ett tecken i B$.
Rad 40 upprepar. När upprepningen är klar innehåller A$ och B$ 255 tecken vardera.
Rad 50 skriver ut samtliga 510 tecken.

Efter programkörningen kan du kontrollera längden av `A$` och `B$` med funktionen `LEN`.



Figur 23: Resultat efter körning av program som skriver ut 510 tecken med kommandot PRINT.

Funktioner för textmanipulation

Dessa funktioner kan användas för att analysera eller manipulera en textsträng: `LEN`, `LEFT$`, `RIGHT$` och `MID$`.

Funktionen `LEN` svarar på hur lång en sträng är (antal tecken). Det kan användas för att kontrollera huruvida en sträng har data i sig, eller om en operation som förlänger en sträng kommer att lyckas eller ej, som här, där man vet att en operation ökar en strängs längd med 30 tecken:

```
10 A$=A$+"AAAAABBBBBCCCCCDDDDDEEEEEFFFFF"
20 IF LEN(A$)<=255-30 GOTO 10
30 PRINT A$
```

Variabeln `A$` kommer att innehålla 240 tecken efter körning. Eftersom en 240 tecken lång sträng inte kan förlängas med ytterligare 30 tecken, ser vi till att testet i det läget utvärderas till falskt och ingen upprepning sker.

`LEFT$` och `RIGHT$` skickar tillbaka önskat antal tecken från en textsträngs start respektive slut. Om man vill att användaren ska göra ett val, kanske man nöjer sig med att läsa första tecknet. I detta exempel frågar vi användaren han vill avsluta programmet. Vilket svar som helst som inleds med bokstaven `J` betraktas som ett accepterande.

```
10 PRINT "AVSLUTA (J/N)";
20 INPUT A$
30 IF LEFT$(A$,1)<>"J" GOTO 10
```

Notera att frågan på rad 10 inte avslutas med frågetecken, eftersom INPUT skriver ut ett sådant. Det är också därför jag vill avsluta hela programsatsen med ett semikolon – annars kommer frågetecknet på raden efter.

Ett annat användningsområde för `LEFT$` och `RIGHT$` skulle kunna vara ett textäventyr, alltså ett spel där användaren ger kommandon i formatet `[VERB] [SUBSTANTIV]`, t.ex. `ÖPPNA DÖRR`. Följande kod kontrollerar om användaren vill *ta nyckeln* – notera placeringen av blanksteg:

```
10 INPUT A$
20 IF LEFT$(A$,3)="TA " AND
   RIGHT$(A$,8)=" NYCKELN"
   THEN PRINT "DU VILL TA NYCKELN"
30 GOTO 10
```

Detta sätt att analysera det användaren skriver ignorerar eventuella mellanord. På engelska skulle en sådan här kod kunna identifiera att både `GET KEY`, `GET THE KEY`, `GET SKELETON KEY` och `GET THE SEKELETON KEY` identifieras som önskemålet att ta nyckeln.

Slutligen har vi `MID$` som fungerar ungefär som `RIGHT$` fast med ett extra läge: Om man vet vilken position och stränglängd man vill titta på, kan man ange strängen, startpositionen och längden som argument. `MID$` svarar då med vad som finns på positionen i fråga, med den angivna längden.

Använder man `MID$` med två argument, så plockas tecknen från höger, fast positionen räknas fortfarande från vänster.

Tabellen ger en översikt:

Exempel	**Resultat**	
`LEFT$("SVEN",2)`	`SV`	*Tecken 1 och två från vänster.*
`RIGHT$("SVEN",2)`	`EN`	*Tecken 1 och två från höger.*
`MID$("SVEN",2)`	`VEN`	*Innehållet till höger om andra tecknet från vänster.*
`MID$("SVEN",1,2)`	`VE`	*Andra tecknet från vänster och totalt två tecken.*

Hämta input från användaren

Din dator har flera sätt att ta emot information, eller input, från användaren. Input kan till exempel komma från tangentbordet, från en mus, en styrspak eller från en ljuspenna. Just Commodore BASIC 2.0 second release kan bara ta emot input från tangentbordet, men annan input kan läsas av genom att använda `PEEK`.

Kommandot `INPUT` låter användaren skriva in en text, ett heltal eller ett flyttal genom att skriva på tangentbordet och trycka **Return** när inmatningen är klar. Detta gör man typiskt när man vill fråga användaren om något. `GET` däremot, berättar för dig vilken tangent som är nedtryckt just nu, och är därför kanske mer lämpad för realtidsstyrning. Här följer ett enkelt exempel på användning av `INPUT`:

```
10 INPUT "VAD HETER DU";A$
20 PRINT "HEJ " A$
```

Resultatet av en körning av detta program skulle kunna se ut så här:

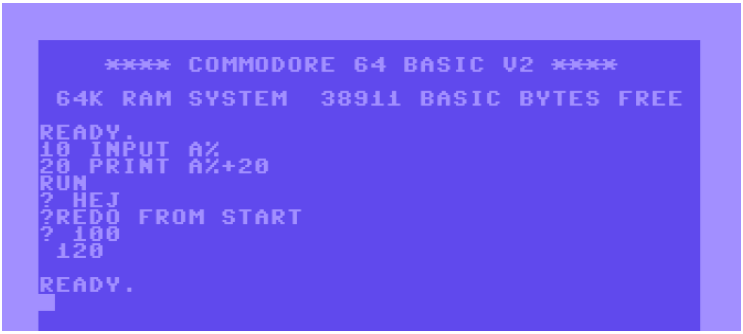
```
VAD HETER DU? KLAS
HEJ KLAS

READY.
```

Kom ihåg att `INPUT` automatiskt lägger på frågetecknet.

I händelse av att `INPUT` förväntar sig en inmatning i ett specifikt format, som på bilden nedan där ett heltal förväntas, så frågar datorn igen om givet data inte uppfyller kriterierna.

Se bild på nästa sida.



Figur 24: Vid felaktig inmatning frågar datorn igen.

Till skillnad från `INPUT` ger `GET` information om vilken tangent på tangentbordet som är nedtryckt just nu, typiskt en sträng som antingen är tom (om ingen tangent är nedtryckt) eller innehåller ett tecken (som representerar den nedtryckta tangenten). `GET` kan användas med vilken variabeltyp som helst, men det är egentligen inte vettigt att använda den med någon annan typ än sträng (t.ex. `A$`). Om den används med en heltalsvariabel, kommer svaret att vara `0` om ingen tangent är nedtryckt eller om tangenten **0** är nedtryckt, om **1** till **9** är nedtryckt blir svaret motsvarande `1` till `9`, men om någon annan tangent trycks ner uppstår felet *syntax*.

Typiska användningsområden för `GET` är att man vill att användaren ska trycka på en tangent för att fortsätta, eller rutiner för att låta en användare styra ett datorspel med tangentbordet. I detta exempel, måste användaren trycka på **Space** (blankstegstangenten) för att fortsätta:

```
10 PRINT "TRYCK BLANKSTEG FÖR ATT FORTSÄTTA"
20 GET A$
30 IF A$<>" " THEN 20
40 PRINT "NU FORTSÄTTER VI"
```

Först när **Space** (blanksteg) tryckts ner, fortsätter programmet.

Omvandla mellan text och tal

Funktionen `CHR$` omvandlar en PETSCII-kod till sitt motsvarande tecken. Funktionen `ASC` gör det motsatta: Den tar en ett tecken och ger dess PETSCII-kod till svar. Givet att `CHR$(65)` ger `"A"` och `ASC("A")` ger `65` så ger `ASC(CHR$("65"))` svaret `65` och `CHR$(ASC("A"))` svaret `"A"`. Dessa funktioner kan vara användbara för den som sparar tecken i bytes, t.ex. genom att använda `PEEK` och `POKE`.

Om man däremot vill veta vilket tal som en textsträng representerar, används funktionen `VAL`. Läser man av `VAL("45")` får man det numeriska värdet `45` som svar. Det motsatta behöver sällan göras, för numeriska värden kan generellt sett skrivas ut som text, men i de situationer man behöver just detta så finns `STR$`. Anropet `STR$(61)` ger textvärdet `"61"` som svar. Om du t.ex. vill kontrollera hur ett värde representeras i datorns minne kan detta vara intressant, då strängen `61` lagras som `54` (tecknet `"6"`) och `49` (tecknet `"1"`).

Textminnet

Istället för att använda `PRINT` till att skriva ut text på skärmen, kan textminnet manipuleras direkt med `POKE`. Commodore 64 har ett textminne på 1000 tecken (40×25). Minnesarean är från `1024` till och med `2023`. Genom skriva till dessa adresser, kan du direkt manipulera vilka tecken som ska visas på skärmen. För att skriva tecknet `A` högst uppe till vänster, skriv:

```
POKE 1024,1
```
(endast Commodore 64)

För att skriva tecknet `B` längst nere till höger, skriv:

```
POKE 2023,2
```
(endast Commodore 64)

Värdet som skrivs styr vilket tecken som hamnar på skärmen. Bokstäverna `A` till `Z` har värde `1` till `26`. Om gemener används (se avsnittet om skiftläge först i detta kapitel) har versalerna värde `65` till `90`. För reverserade tecken, addera `128` till värdet.

Om du använder VIC-20 har du endast 506 tecken på skärmen (22×23). Minnesarean varierar beroende på om du har en expanderad VIC-20 eller ej, men på en icke expanderad VIC-20 finns textminnet på `7680` till och med `8185`. Du får dock ingen synlig effekt genom att skriva text där, om du inte antingen byter ut ett befintligt tecken, eller dessutom manipulerar samtidigt manipulerar teckenfärgminnet (`38400` till och med `38905`).

Så, för att lösa motsvarande uppgift på VIC-20, används följande kod för att skriva tecknet `A` högst uppe till vänster (endast VIC-20):

```
POKE 7680,1
POKE 38400,150
```

För att skriva tecknet B längst nere till höger, skriv (endast VIC-20):

```
POKE 8185,2
POKE 38905,150
```

KAPITEL 13: GRAFIK

Grafik

Commodore BASIC 2.0 second release har inga särskilda kommandon för pixel-grafik. Många funktioner som din dator erbjuder, kommer man åt genom att skriva olika värden till olika minnesadresser. Det innebär att man använder kommandot `POKE` väldigt ofta när man jobbar med grafik på en VIC-20 eller Commodore 64.

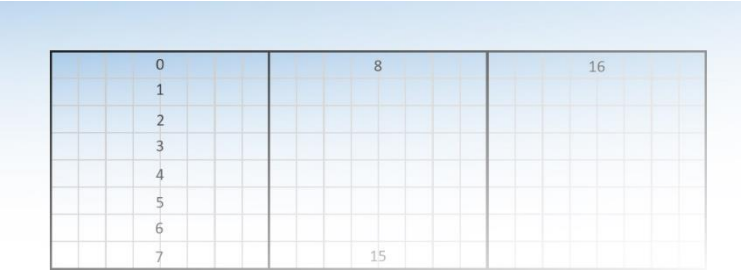
Detta kapitel ger en kort introduktion till din dators kapacitet att visa grafik, med fokus på den lite mer kapabla datorn Commodore 64, som också har stöd för sprites. Dessutom introduceras konceptet med teckenbaserad grafik, som är tillgängligt på båda datorerna. Innehåll:

- Introduktion
- Grafik på Commodore 64
- Grafik på VIC-20
- Teckenbaserad grafik
- Flerfärg
- Sprites på Commodore 64

Det kan vara bra att hålla koll på datorns färgpalett, som anger vilken färg som har vilken kod på din dator. Både färgpaletten för Commodore 64 och färgpaletten för VIC-20 finns listad i kapitlet om metoder, under avsnittet om bakgrundsfärg och borderfärg.

Introduktion

De båda datorerna är aningen olika konstruerade när det gäller kapacitet och minnesmodell men i båda fallen finns det två koncept för att skapa grafik. **Teckengrafik**, som ofta är att föredra när det handlar om rörlig grafik, bygger på att man ändrar på datorns teckensnitt och monterar bilder med de tecken man har skapat. Sen har Commodore 64 stöd för s.k. **bitmapsgrafik**, som passar bättre för stillbilder. Bilder skapas genom att man tänker på skärmens alla pixlar som grupper om 8×1 stycken i en horisontell rad, där varje sådan grupp har en minnesadress. Den binära representationen av talet berättar vilka pixlar inom gruppen som ska vara tända eller släckta. Nästa minnesadress beskriver raden under föregående och när åtta sådana grupper är passerade, är nästa grupp placerad till höger om föregående område på 8×8 pixlar. Bilden visar ett exempel där siffrorna indikerar ordningen.



Figur 25: Ordningen för varje grupp av åtta pixlar som delar minnesadress. Rektanglarna representerar ett färgområde (se nedan).

Textminnet påverkar bildens färger. Befintlig text "lyser igenom" till bitmapsgrafiken och påverkar bildens förgrunds- och bakgrundsfärg för varje tecken. En högupplöst bild som inte är monokrom (eller en lågupplöst bild som består av fler än fyra färger) måste alltså *både* ha pixeldata och färgdata, där färgerna lagras som text. Bilden ovan visar ett färgområde som svart kvadrat om 8×8 pixlar.

Grafik på Commodore 64

Innan man sätter igång behöver man konfigurera var i minnet bildens pixlar (en s.k. *bitmap*) och färgerna (*color map*) ska ligga lagrade. Grafikminnet kan hantera 16 K data, och datorn behöver veta vilka 16 K som grafikminnet ska använda.

Adress `53272` har flera användningsområden. I bitmapsläge anger den adressen till färginformationen (hög nibble) och adressen där pixeldata finns. Värdet 0 på bit 3 anger adress 0, värde 1 anger adress `8192` för pixeldata. Kodraden `POKE 53272,25` konfigurerar alltså grafikminnet.

Därefter ska grafikläget aktiveras, vilket görs genom att sätta bit 5 på adress 53265 enligt `POKE 53265,PEEK(53265) OR 32`. Så med dessa två kodrader på plats, är vi redo att börja jobba med bitmapsgrafik.

```
10 POKE 53272,25
20 POKE 53265,PEEK(53265) OR 32
```

Därefter är det helt enkelt fritt fram börja jobba med grafiken. Färgerna ligger nu på adress `1024` och pixeldata på adress `8192`. Låt oss göra en liten glad gubbe - en *smiley* - högst upp i vänstra hörnet av skärmen. Först vill vi ha svart botten och gula pixlar. Varje tecken har bakgrundsfärgen i sin låga nibble (som ska vara 0) och förgrundsfärgen i sin höga nibble (som ska vara 7, eftersom 7 är gul). För att få in 7 i hög nibble, multiplicerar man det med 16. Inget mer behöver adderas, eftersom svart är 0. Detta gör tecknet högst uppe till vänster svart med gul förgrundsfärg:

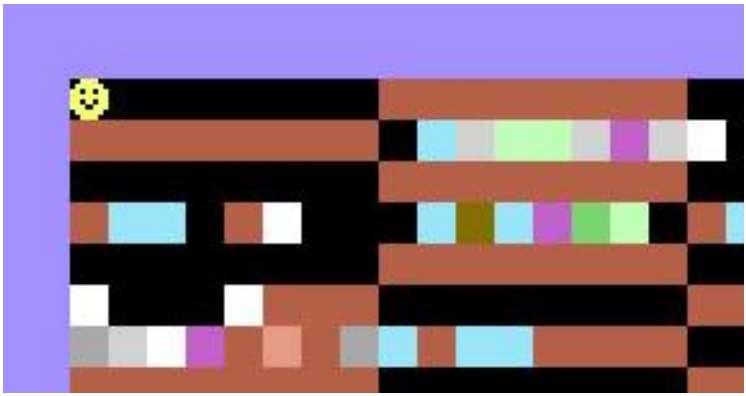
```
30 POKE 1024,112
```

Anledningen till att gul förgrund mot svart botten har värdet 112 beror på att bakgrundsfärgen lagras i låg nibble (vilket innebär att den är exakt som anges i färgtabellen i kapitlet om metoder) och att förgrundsfärgen anges i hög nibble. Förgrundsfärgen måste således multipliceras med 16 och adderas till bakgrundsfärgen. Gul har färgkod 9, 1×16 är 112 och 112+0 (där 0 är färgkoden för bakgrundsfärgen) ger 112.

Att slutligen ange vilka pixlar som ska vara tända eller släckta för varje grupp om 8×1 pixlar är enkelt. Man anger helt enkelt den decimala representationen av det 8-bitarstal som motsvarar tända (1) och släckta pixlar på området. Dessa kodrader representerar en smiley (alltså en gul glad gubbe):

```
40 POKE 8192,60
50 POKE 8193,126
60 POKE 8194,219
70 POKE 8195,255
80 POKE 8196,219
90 POKE 8197,231
100 POKE 8198,126
110 POKE 8199,60
```

Så här kan resultatet se ut:



Figur 26: En smiley skapad med en bitmap och en colormap.

Runt om ser vi inga satta pixlar, för att minnesarean råkar innehålla 0. Men vi ser en massa färger, och det beror på att textminnet innehåller oönskad data och det enda området vi har satt är det högst uppe till vänster, under smileyn. En kortversion av programmet skulle kunna se ut så här:

```
10 POKE53272,25
20 POKE53265,PEEK(53265) OR 32
30 POKE1024,112
40 FOR A=8192 TO 8199:READ B:POKE A,B:NEXT
50 DATA 60,126,219,255,219,231,126,60
```

För den som vill utforska högupplöst bitmapsgrafik, följer här en implementation av Lorenz-attraktorn för Commodore 64[28].

```
10 REM SKYDDA BASIC
20 POKE 56,32:POKE 52,32
30 REM PIXELGRAFIK
40 CLR:POKE 53272,25
50 POKE 53265,PEEK(53265) OR 32
60 PRINT "{CLEAR}"
70 REM RENSA BEFINTLIGA PIXLAR
80 POKE 88,0:POKE 89,63
90 POKE 113,64:POKE 114,31
100 POKE 12,1:SYS 45760
110 REM LORENZ
120 X=5
130 Y=5
140 Z=5
150 T=0
160 S=1/200
170 D=10
180 R=28
190 B=8/3
200 T=T+0.1
210 DX=D*(Y-X)
220 X1=X+DX*S
230 DY=(R*X-Y)-X*Z
240 Y1=Y+DY*S
250 DZ=X*Y-B*Z
260 Z1=Z+DZ*S
270 X=X1
280 Y=Y1
290 Z=Z1
300 REM POSITION
310 XX=150+4*X:YY=20+3*Z
320 REM RITA PIXEL
330 ROW=INT(YY/8):CH=INT(XX/8)
340 LINE=YY AND 7:BIT=7-(XX AND 7)
350 BYTE=8192+ROW*320+CH*8+LINE
360 POKE BYTE,PEEK(BYTE) OR 2^BIT
370 REM UPPREPA
380 IF T<1000 GOTO 200
```

Så här fungerar programmet:

[28] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/lorenz.bas

Rad 20 modifierar pekarna som anger variabelminne och BASIC-minne. Rad 40 rensar variabler och `DATA`-pekare och aktiverar läget för högupplöst grafik på Commodore 64. Rad 80-100 rensar skärmen från befintlig grafik. Rad 120-290 utgör formeln för Lorenz-attraktorn och rad 310 räknar ut den pixel på 320×200-matrisen som ska sättas - den koden är identisk med för Commodore 128-versionen (appendix D). Slutligen ansvarar rad 330-360 för att omvandla en X- och Y-koordinat till en minnesadress och ett bitmönster, som sedan sätts (se nedan). Rad 380 upprepar.

En mer kommenterad version av detta program för Commodore 128 finns i appendix D, men VIC-20 saknar möjligheten att visa bitmapsgrafik.

Själva konverteringen till en X- och Y-koordinat är relevant för att algoritmen som ritar Lorenz-kurvan räknar fram en punkt på en matris på 320×200 pixlar som ska sättas, vilket Commodore 128 erbjuder programmeraren genom sin BASIC-tolk. Men Commodore 64 har en annan ordning på pixlarna som bryter rad var 8:e pixel, och sedan hoppar fram 8 pixlar efter 8 radbryten. Den som ska sätta en pixel på en specifik punkt på skärmen på en Commodore 64, måste först räkna ut vilken 8×8 pixlar stor cell som punkten ska landa i, sedan på vilken av de åtta raderna däri, för att slutligen sätta en bit på rätt position på den raden.

När vi går in i att utföra den uppgiften har vi lagrad den önskade X- och Y-koordinaten i variablerna `XX` respektive `YY`, sen sker magin på raderna 330-360. Nu ska positionen omvandlas till 8×8-celler och en position inom den cellen.

På rad 330 räknar vi ut vilken rad av 8×8-celler som avses, och vilken cell på den raden.

Rad 340 räknar ut vilken rad inom cellen som avses, samt vilken bit på den raden som är aktuell att sätta.

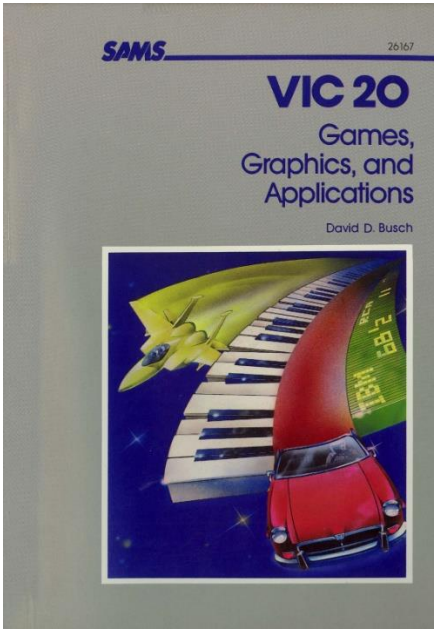
Rad 350 sätter den aktuella biten, utan att påverka några andra bitar på raden.

Därmed har vi åstadkommit ett av de ting som Commodore 128-kommandot `DRAW` kan utföra, nämligen att tända en pixel på en önskad koordinat på 320×200-matrisen.

Grafik på VIC-20

VIC-20 har egentligen inte stöd för bitmapsgrafik, men datorn har olika grafiklägen och kan också visa högupplöst grafik (då 176×184 pixlar) eller flerfärgsgrafik (88×184 pixlar). Strategin man måste använda är den som nämndes tidigare, som lämpar sig för rörlig grafik, alltså teckengrafik. Men man har möjlighet att använda egna symboler i stället för datorns inbyggda tecken.

Om du vill utforska VIC-20 genom exempel, rekommenderas boken "VIC-20 Games, Graphics and Applications" av David D. Busch.



Figur 27: En bok som lär dig programmera VIC-20 genom exempel.

Teckenbaserad grafik

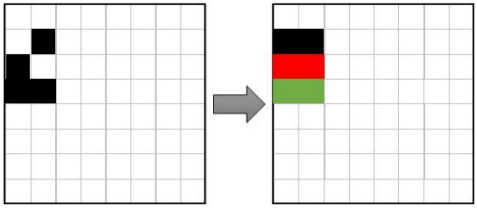
Commodore BASIC 2.0 second release har stöd för teckenbaserad grafik, och detta fungerar lika dant oavsett om du sitter vid en VIC-20 eller en Commodore 64, även om utseendet är lite annorlunda. Möjligheten att mata in teckenbaserad grafik i en sträng aktiveras av att man skriver ett citattecken (`"`) och avaktiveras nästa gång ett citattecken skrivs. Detta kräver disciplin och korrekta knapptryckanden. Första kapitlet av boken *Grafik och ljud på VIC-64*[29] [sic] av Sune Windisch beskriver detta, och det som sägs är sant även för VIC-20.

Flerfärg

Det finns flera grafiklägen att välja på, och din dator kan visa totalt fyra färger per 8×8-yta i flerfärgsläget. Detta sker på bekostnaden av pixlarnas storlek, så egentligen finns det då bara 4×8 logiska pixlar på ett område av 8×8 pixlar.

Datorn har bara avsatt en enda bit per bildelement (eller pixel), så hur kan den visa fyra olika färger? Genom att gruppera pixlarna i par så ändrar man två tänkbara kombinationer (0 eller 1) till fyra tänkbara kombinationer (00, 01, 10 eller 11).

Bakgrundsfärgen och den första förgrundsfärgen kontrolleras då av (motsvarande) teckenminnet, medan övriga färger (multifärg 1 och multifärg 2) är fasta för hela bilden.



Figur 28: Högupplöst pixelgrafik (till vänster) och flerfärgsgrafik (till höger).

För exempel, se manualen för respektive dator.

[29] För att spela an på VIC-20 så marknadsfördes Commodore 64 till en början som VIC-64 i Sverige, vilket gör att viss svenskproducerad dokumentation benämner Commodore 64 som just VIC-64.

Sprites på Commodore 64

Ordet *sprite* betyder ungefär hustomte eller fe, vilket inte är helt orimligt givet att en sprite är ett stycke grafik som kan röra sig fritt över skärmen. Vissa datorer har sprites inbyggt i hårdvaran (t.ex. Commodore 64 och Commodore 128) vilket är bäst - det ger bäst prestanda och det är mycket enklare att skriva datorprogram som använder sprites om datorn redan från början har stöd för sprites. På andra system (t.ex. VIC-20 eller ZX Spectrum) måste den som vill använda sprites tillhandahålla en mjukvarulösning, vilket kan vara svårt och kan bli långsamt.

Storleken på en sprite, 24 pixlar bred och 21 pixlar hög, men det finns en förklaring till det. Varje byte beskriver 8 pixlar, och en sprite är tre bytes bred. 3 gånger 21 blir 63, vilket innebär att en hel sprite ryms inom 64 bytes. Trots att bara 63 bytes behövs, ligger varje bild lagrad med 64 bytes mellanrum i minnet.

Commodore 64 kan visa åtta sprites, numrerade från 0 till 7.

Slå på och av en sprite

De åtta bitar som finns på adress `53269` påverkar var sin sprite. Bit 0 påverkar sprite 0, bit 1 påverkar sprite 1, och så vidare. Eftersom den tredje biten, bit 2, är värd 4, kommer alla sprites utom sprite 2 att släckas om `4` skrivs till adress `53269`. Det är inte helt säkert att du ser någon effekt av detta, för en sprite kan ligga utanför det synliga området på skärmen och en sprite skulle kunna sakna bildmotiv, och således vara helt transparent.

```
POKE 53269,4
```

Så händer det inget, läs bara vidare.

Om du vill slå av och på sprites individuellt, tänk då på att maska ut den bit som ska sättas eller nollas. Se avsnittet om bitmaskning i kapitlet om metoder, tidigare i denna bok.

Positionera en sprite

Varje sprite har en minnesadress för sin X-koordinat och sin Y-koordinat. Position 0,0 ligger högst upp till vänster i bild, men utanför det synliga området. Detta för att en sprite ska kunna glida in i synfältet. Sprite 0 har sin X-koordinat på adress `53248` och sin Y-koordinat på `53259`, sprite 1 har sin X-koordinat på adress `53250` och sin Y-koordinat på `53251`, och så vidare.

Eftersom jag har aktiverat sprite 2, är det `53252` och `53253` som är intressant för mig.

```
POKE 53252,50
POKE 53253,70
```

Nu kan du flytta runt dina sprites över skärmen. Men om du sätter X-koordinaten (i mitt fall `53252`) till `255`, så har du ändå inte lyckats nå skärmens högra kant. Och större tal än `255` går inte att sätta på en minnesadress. För detta har vi en speciell funktion på adress `53264`, alltså direkt efter alla X- och Y-koordinater. Här har varje sprite en flagga som anger om de har passerat position `255` eller ej. Eftersom det är just en flagga, så måste bitmaskning användas även här (precis som när en sprite slås på och av).

Detta placerar sprite 2 på position `256` (utan hänsyn till övriga bitar på adress `53264`):

```
POKE 53264,4
POKE 53252,0
```

Ändra motivet på en sprite

En sprite är 63 bytes stor, eftersom den innehåller tre bytes per rad i 21 rader. Den skulle inte kunna vara 64 bytes stor eftersom 64 inte är jämt delbart med 3. Trots detta konsumerar en sprite 64 bytes minne, för de lagras i minne som avsätts genom att man pekar ut 64 bytes stora sektioner. Till detta har vi en pekare per sprite per sprite på adresserna `2040`-`2047`.

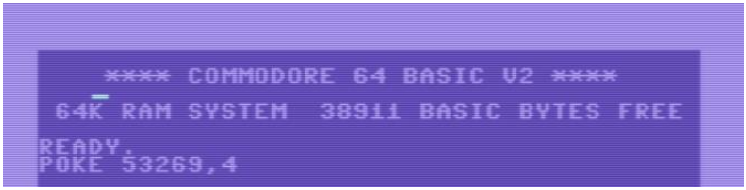
För att välja ut en adress, dividera din tänkta adress med just 64 och skriv in värdet i respektive pekare. Om jag vill att den tredje spriten ska ha bilddata på adress `8320` så lagrar jag `130` på adress `2042`.

```
POKE 2042,130
```

Varför `2042`? För att det är den tredje pekaren av de åtta pekare som anger adressen för bilddata. Varför `130`? För att det är just `130` som blir `8320` när det multipliceras med `64`.

Det är lätt att se hur man kan skapa animationer genom att med jämna mellanrum skriva till adresserna `2040`-`2047`.

Har du gjort dessa steg, kan du börja fylla på med bilddata på adressen som du pekat ut. Ett åtta pixlar långt och två pixlar tjockt streck borde dyka upp på skärmen om du skriver `255` till adress `8320` och `8323`.



Figur 29: En sprite med en två pixlar tjock horisontell linje.

Mer att ta reda på

En sprite kan ha olika egenskaper, färger och storlek, vilket beskrivs i den tidigare nämnda boken *Grafik och ljud på VIC 64*.

KAPITEL 14: LJUD

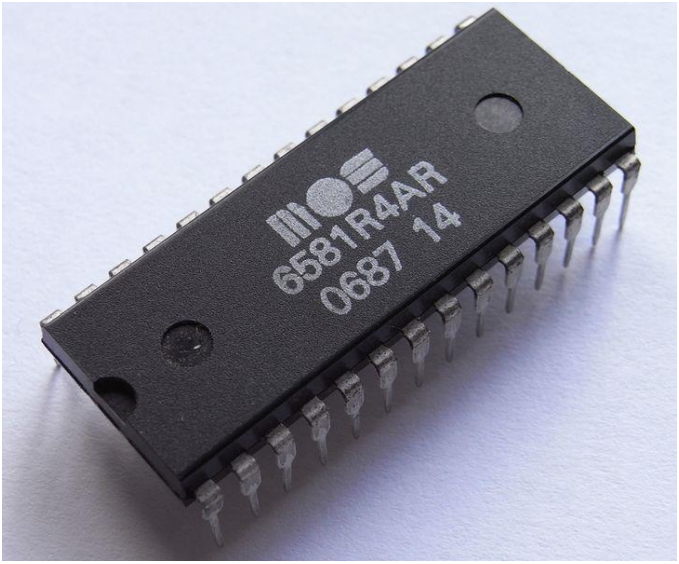
Ljud

Både Commodore 64 och VIC-20 innehåller en synthesizer, där den som sitter i Commodore 64 är lite mer avancerad. Precis som när det gäller grafik, finns inga dedikerade BASIC-kommandon för att använda ljudet. Det handlar även denna gång om att skriva till minnet.

VIC-20 har tre röster som var och en kan spela tre oktaver. Den första har ett register från oktav 1 till 3, den andra från 2 till 4 och den tredje från 3 till 5. Dessutom kan VIC-20 spela brus.

Tack vare SID-chippet har Commodore 64 tre röster som kan spela toner i olika vågformer med stöd för variabel pulsbredd och filter.

Så eftersom Commodore BASIC 2.0 second release inte har några kommandon för ljud, och eftersom ljud fungerar lite olika i dessa maskiner, är kapitlet uppdelat i två delar. Exemplet på nästa sida fungerar endast på Commodore 64, och i nästa del visas en introduktion till att spela ljud på VIC-20.



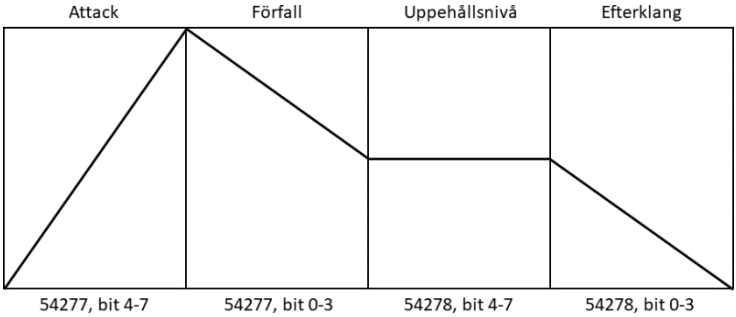
Figur 30: SID-chippet i en Commodore 64.

Ljud på Commodore 64

Inledningsvis bör man alltid rensa hela SID-registret, som är lokaliserat mellan `54272` och `55295` (`D400` - `D7FF` i hexadecimal). Detta gör man för att undvika att eventuella tidigare ljudeffekter eller musik ska påverka ljudet man vill spela.

```
FOR A=54272 TO 55295:POKE A,0:NEXT
```

Den första röstens omslag (attacklängd, förfalloländ, uppehållsnivå och efterklangsländ) konfigureras på adress `54277-54278`. De övriga två ligger på `54284-54285` och `54291-54292`. Den första bytens höga nibble är attack, dess låga är förfall. Den andra bytens höga nibble är uppehållsnivå, dess låga är efterklang.



Figur 31: Omslag för röst 1 av 3 (Commodore 64).

Genom att sätta värde 14 (vars binära representation är `00001110`) på adress `54277`, har jag sagt att jag vill ha 0 som attacklängd och 14 (av 15) som förfallolängd. Då jag tidigare tömde SID-registret, vet jag att både uppehållsnivå och efterklang (`54278`) är 0.

```
POKE 54277,14
```

De fyra låga bitarna på adress `54296` anger volymen, så värdet 15 på den adressen sätter volymen till max.

```
POKE 54296,15
```

Slutligen ska en frekvens väljas. För röst 1 är `54272` den låga byten och `54273` den höga. För röst två är det `54279-54280` och för röst tre är det `54286-54287`.

Frekvens 5632 är ungefär ett E. 16-bitarstalet 5632 har bitmönstret `0001 0110 0000 0000`. Den höga byten är `00010110`, alltså 22, den låga är 0. Då jag nollställt SID-registret, räcker det att ange den höga byten.

```
POKE 54273,22
```

För att börja spela en ton sätter man bit 0 till 1 på adress `54276`. Biten måste vara 0 sedan tidigare, annars har tilldelningen in effekt. De övriga bitarna används för bland annat ringmodulation och vågform. Bit 5 anger sågtand. Alltså, `00100001`, vilket är 33, startar en sågtandston på den första rösten. Bit 0 på adress `54276` måste återställas till 0 för att nästa ton ska kunna spelas.

```
10 POKE 54276,33
20 FORI=0 TO 200:NEXT
30 POKE 54276,32
40 FORI=0 TO 200:NEXT
50 POKE 54276,33
```

Rad 10 spelar en ton. Givet konfigurationen tidigare, ett E.
Rad 20 ger en kort fördröjning.
Rad 30 stoppar tonen.
Rad 40 ger en kort fördröjning.
Rad 50 spelar tonen igen.

När programmet startas kommer du att höra två toner efter varandra. Ett E i med vågformen sågtand. Om programmet startas igen kommer inte första tonen att höras. Det beror på att programmet avslutas med bit 0 på `54276` satt till 1, vilket innebär att rad 10 således inte har någon effekt andra gången.

För mer information om ljudprogrammering på Commodore 64, se användarmanualen för Commodore 64, kapitel 7.

Ljud på VIC-20

VIC-20 har inget dedikerat ljudchip. Det är videochipet (MOS 6560 för NTSC eller MOS 6561 för PAL) som sköter ljudet. Följande exempel fungerar endast på VIC-20.

På VIC-20 måste man tänka på att olika röster har olika frekvensomfång, där den första är lägst och den sista (3:e) är högst. Volymen sätts genom att tilldela ett värde till den låga nibbeln på adress `36878`.

```
POKE 36878,15
```

Att spela en ton på VIC-20 är enklare än på Commodore 64, men man har inte samma flexibilitet. I stället för att bara sätta en flagga, så anger man helt enkelt en frekvens (128 till 255) på en ton man vill spela.

Ljudregister finns på adress `36874` till `36878`. Den första rösten har adress `36874`, den andra har `36875` och den tredje har adress `36876`. För att starta en ton sätter man ett värde (frekvensen) i respektive adress, ju högre värde desto högre frekvens har tonen, undantaget `255` som ger röstens lägsta frekvens.

Det talen mellan 128 och 255 har gemensamt, är att bit 7 är satt till 1. Det är alltså bit 7 som avgör om en röst är av eller på, medan de andra bitarna, bit 0 till 6, anger frekvensen.

Följande program spelar två toner efter varandra på den första (lägsta) rösten.

```
10 POKE 36878,15
20 POKE 36874,210
30 FORI=0 TO 200:NEXT
40 POKE 36874,0
50 FORI=0 TO 200:NEXT
60 POKE 36874,210
70 FORI=0 TO 200:NEXT
80 POKE 36874,0
```

Rad 10 sätter volymen till max.
Rad 20 spelar en ton (ungefär ett G).
Rad 30 ger en kort fördröjning.
Rad 40 stoppar tonen i röst 1 av 3.
Rad 50 ger en kort fördröjning.
Rad 60 spelar en ton i samma frekvens som tidigare.
Rad 70 ger en kort fördröjning.
Rad 80 stoppar tonen.

För mer information om ljudprogrammering på VIC-20, se användarmanualen för VIC-20, kapitel 5.

KAPITEL 15: FELMEDDELANDEN

Felmeddelanden

Commodore BASIC 2.0 second release har sexton felmeddelanden som kan presenteras för användaren i syfte att informera om felaktigheter i programmering, i exekvering av kommandon och övrig användning. Några som du har god chans att stöta på när du följer exemplen i denna bok är *bad subscript*, *can't continue*, *file not open*, *illegal direct*, *illegal quantity*, *next without for*, *out of data*, *out of memory*, *overflow*, *redo from start*, *return without gosub*, *string too long*, *syntax*, *type mismatch*, *undefined function* och *undefined statement*. De förklaras här, i turordning.

Bad subscript

Felet *bad subscript* handlar om att ett felaktigt index har använts. Du kan t.ex. försöka använda element 10 i en array med endast 5 element. Felet kan även uppstå när en funktion som använder en array anropas med en felaktig parameter, t.ex. `CHR$`.

Vad gör jag? Du har förmodligen ett logiskt problem att ta hand om. Vid något tillfälle försöker du skriva till eller läsa från ett element som inte finns för att dess index inte finns representerad i programmet. Kontrollera att allt som används också är deklarerat!

Can't continue

Kommandot `CONT` kan endast användas efter att ett program har stoppats, till exempel genom ett tryck på knappen **Run Stop**. Om `CONT` anropas före programkörning eller efter helt fullföljd programkörning vidtar inte datorn någon åtgärd alls, men om programmet t.ex. har modifierats efter senaste körningen så inträffar felet *can't continue*. Om felet inträffar, starta igen med `RUN`.

Vad gör jag? Starta felsökningen från början med `RUN`.

File not open

Så snart ett kommando (till exempel `CMD` eller `GET#`) försöker använda en kanal som inte först är öppnad med `OPEN` inträffar felet *file not open*.

Vad gör jag? Anropa `OPEN` först.

Illegal direct

Vissa kommandon (till exempel `DEF`, `GET` eller `INPUT`) får endast användas i ett program, inte i direktläge. Om man försöker använda dem i direktläge uppstår felet *illegal direct*.

Vad gör jag? Skriv dessa kommandon i ett program, inte i direktläget.

Illegal quantity

När helst ett värde är för stort eller för litet för vad som förväntas i ett visst sammanhang uppstår felet *illegal quantity*. Det skulle kunna vara när ett tal större än 32767 i en heltalsvariabel eller när du försöker öppna en fil med ett filnummer större än 255. Saknas begränsning på hur stort tal som förväntas, kommer felet att uppstå om det givna talet är för stort för datorn att hantera.

Vad gör jag? Kontrollera att dina iterativa formler inte skenar, memorera maxgränsen för värden i olika situationer.

Next without for

Detta fel inträffar när din dator påträffar en `NEXT`-sats som inte hör till den *senast* påträffade `FOR`-satsen.

Vad gör jag? Den `FOR`-sats som öppnas sist ska stängas först och antalet `NEXT`-satser ska gå jämt ut med antalet `FOR`-satser.

Out of data

Felet inträffar när datorn påträffar en `READ`-sats, trots att alla `DATA`-satser redan är lästa (förhindras med `RESTORE`) eller när en `READ`-sats påträffas utan att några `DATA`-satser finns.

Vad gör jag? Kontrollera att det finns ett `DATA`-värde för varje anrop på `READ`, och använd `RESTORE` om så behövs.

Out of memory

Oftast beror troligen på en felaktig inläsning från något sekundärminne, och troligen datasette (kassettband). Om en pekare tolkas som att ligga utanför tillgängligt RAM-minne, inträffar felet *out of memory*.

Vad gör jag? Om du t.ex. bygger en array, kontrollera att den faktiskt får plats i minnet. Annars, gör en hårdvarudiagnostik.

Overflow

Felet *overflow* uppstår när ett värde inte får plats där det ska lagras, t.ex. ett tal är för stort för variabeln den ska lagras i. Felet uppstår även när ett negativt tal används tillsammans med `ON...GOTO` eller `ON...GOSUB`.

Vad gör jag? Se till att din variabel förmår att lagra det värde du vill spara, och kontrollera att den subrutin du försöker anropa faktiskt finns.

Redo from start

Felet *redo from start* bryter inte programmet, utan pausar programmet till dess att användaren har levererat en korrekt input. Ett typiskt fall är en `INPUT`-sats som förväntar sig ett tal med får en textsträng av användaren.

Om ett heltal förväntas och ett flyttal ges, antas närmsta underliggande heltal avses.

Om et heltal förväntas och en textsträng ges, inträffar *redo from start*.

Dessa situationer kan testas med följande program:

```
10 INPUT A%
20 PRINT A% "MULTIPLICERAT MED TVÅ ÄR" A%*2
```

Vad gör jag? Kontrollera att det är tydligt vad som förväntas av ditt programs användare.

Return without gosub

Felet *return without gosub* uppstår när helst en `RETURN`-sats påträffas utan att `GOSUB`-sats exekverats innan. Alltså, `RETURN` kan inte köras mot en tom *call stack*.

Vad gör jag? Kontrollera att `RETURN` endast förekommer i slutet av en subrutin, inte i något annat flöde.

String too long

Typiskt uppstår felet K när programmeraren försöker lagra fler än 255 tecken i en strängvariabel (alltså en variabel vars postfix är `$`).

Vad gör jag? Kontrollera att ditt program inte innehåller logik som förväntar sig strängar längre än 255 tecken.

Syntax

När helst en inmatning strider mot förväntad satsuppbyggnad inträffar felet *syntax*. Dessutom inträffar felet när en funktion får fel antal argument. Funktionen SGN förväntar sig ett argument, så om man försöker anropa SGN antingen utan några argument eller med två argument så inträffar felet *syntax*.

Vad gör jag? Förmodligen fel.

Type mismatch

Om en textsträng anges när ett tal förväntas eller tvärt om, uppstår felet *type mismatch*.

Vad gör jag? Kontrollera att det är tydligt vad som förväntas av ditt programs användare och att du har koll på hur du hanterar data i ditt program.

Undefined function (eller undef'd function)

Om en funktion som inte definierats med `DEF FN` försöker anropas med `FN` uppstår felet *undefined function*.

Vad gör jag? Kontrollera att användandet av `FN` verkligen sker efter användandet av `DEF FN`.

Undefined statement (eller undef'd statement)

Om du (eller en kodrad i ditt program) försöker hoppa till ett radnummer som inte finns i ditt program, uppstår felet *undefined statement*. Detta görs typiskt, men inte nödvändigtvis, i direktläge med `RUN [RADNUMMER]` eller i runtime-läge med `GOTO` eller `GOSUB`.

Vad gör jag? Kontrollera att funktionen du anropar faktiskt finns definierad, med hänsyn till de olika sätt en funktion kan definieras på.

KAPITEL 16: COMMODORE BASIC 2.0 DOS

Commodore BASIC 2.0 DOS

Diskoperativsystemet (DOS) är implementerat i din diskdrive. Kommandon för att utföra vissa uppgifter finns i Commodore BASIC, och bland dessa hittar man möjligheten att skicka kommandon till diskdriven. För att följa med i detta kapitel behöver du ha tillgång till en diskdrive av modell 1540, 1541, 1570 eller 1571. Dessutom behöver du ha en floppydisk som inte innehåller något som du inte kan avvara, för det som ligger lagrat på den diskett du använder när du följer med i övningarna kommer att försvinna. Alla praktiska frågor kring hur en diskdrive fungerar och hur disketter används finns i din manual.

Detta kapitel är en introduktion som presenterar det du behöver veta för att kunna dra nytta av DOS i ditt arbete i BASIC. För en djupare insikt rekommenderas boken *Inside Commodore DOS: The Complete Guide to the 1541 Disk Operating System*, ISBN 0835930912.

Kapitlet innehåller följande delar:

- Vad finns på disketten?
- Hämta ett program från disk
- Spara ett program till disk
- Verifiera det sparade programmet
- Återkalla en datafil i BASIC

Om du vill spara specifikt minne på disk, t.ex. ett maskinkodsprogram, se avsnittet om detta i kapitlet om metoder.

Vad finns på disketten?

För att se innehållet på den diskett som sitter i diskdriven laddar du innehållet i filen `$` till BASIC-minnet. Alltså `LOAD "$",8`. Här är det värt att ha papper och penna framme, för det innebär att du för det innebär du förlorar det BASIC-program i minnet varje gång du vill titta på disketten. Välj diskett först, skriv nödvändiga anteckningar, börja jobba sen!

(Commodore 1541 levererades med ett program som hette *DOS Wedge*. I det programmet finns ett kommando som heter `@$` som visar innehållet på en diskett utan att skriva över BASIC-minnet, precis som kommandona `CATALOG` eller `DIRECTORY` i nyare versioner av Commodore BASIC.)

Siffran `8` i kommandot `LOAD "$",8` är viktig, för om du inte ändrat enhetsbeteckning eller om du inte har flera diskdrives inkopplade så säger `8` att det är diskdriven som du avser med kommandot.

Om något går fel, kan du behöva *formatera* disketten. Det innebär att din diskdrive skriver ner information om vad disketten kan lagra och att den är tom, så att kompatibla diskettstationer kan läsa disketten. Använd följande kommando:

```
OPEN 1,8,15,"N0:MIN DISKETT,65":CLOSE 1
```

Detta ger disketten titeln `MIN DISKETT` och ID-numret `65` (som visas i högermarginalen av fillistan). Igen avser siffran `8` din diskdrive. Särskilt om du har fler diskdrives är det viktigt att hålla redan på denna. Tänk dig att du har två diskdrives med enhetsbeteckning `8` respektive `9`, och du avser att formatera den ena, men råkar välja den andra! Det kan leda till permanent förlust av data. Efter att ha läst det här kapitlet kommer du att veta vad alla siffror betyder.

Hämta ett program från disk

Kommandot `LOAD` kan hämta ett program både från kassett (se din manual) och från diskett. Vill du hämta från diskett så måste du veta, åtminstone ungefär, vilket filnamn programmet har. Därför är det bra att veta hur man hämtar ut listan av filer (som beskrivs ovan). Om du har konstaterat att filen du vill hämta heter `JUPITER LANDER` så kan du hämta den genom att ladda in `JU*`, givet att `JUPITER LANDER` är det första programmet på disketten vars två inledande bokstäver är `J` och `U`. Tecknet `*` är alltså ett slags joker i filnamnssammanhang. Du kan således hämta det första programmet på disketten genom att begära filen `*`, helt oavsett vad den faktiskt heter.

Även om alla argument som `LOAD` kan hantera rent tekniskt är frivilliga, så måste du som användare av en diskdrive ange två argument: Vad heter filen du vill läsa in och vilken enhet vill du läsa från?

```
LOAD "filnamn", enhet
```

Det tredje argumentet nämndes i kapitlet kommandon. Om det utelämnas, antas 0 avses, vilket betyder att programmet läses in till datorns BASIC-minne. Du läser alltså in dina BASIC-program genom att utelämna det tredje argumentet (eller specificera 0). Om du istället skriver 1 så läses programmet in till den plats i minnet det sparades från. Typiskt läser du in dina maskinkodsprogram genom att skriva:

```
LOAD "MITT PROGRAM",8,1
```

Spara ett program till disk

Program sparas med kommandot `SAVE`. Även här är alla argument rent formellt frivilliga, men för att spara till disk behöver du åtminstone ange filnamn och enhetsnummer. Om du har ett BASIC-program i minnet, en disk med ledigt utrymme i enhet `8` där filnamnet `MITT PROGRAM` är ledigt, sparar du det på följande vis:

```
SAVE "MITT PROGRAM",8
```

Det tredje argumentet har samma syfte som hos `LOAD`: Värdet `0` (eller utelämnat värde, som ovan) sparar BASIC-programmet som finns i minnet och `1` sparar ett maskinkodsprogram. Men som påpekats i avsnittet om att spara minne på disk i kapitlet om metoder, så har inte Commodore BASIC 2.0 second release något stöd för att specificera vilket minne som ska sparas, vilket innebär att man får välja någon annan lösning för att åstadkomma detta - exempelvis någon av de som förslås i kapitlet.

Om det är ett BASIC-program som du har sparat, är nästa steg att validera att programmet blev sparat korrekt.

Verifiera det sparade programmet

Även `VERIFY` tar ett filnamn och ett enhetsnummer som måste anges när man arbetar med BASIC-program på floppydisk. Kommandot kontrollerar att innehållet i filen stämmer överens med innehållet i BASIC-minnet, och således vet vi att ett tidigare försök att spara ett program till fil med `SAVE` har lyckats. `VERIFY` svarar med ett `OK` om innehållet i filen stämmer med innehållet i minnet, annars med felet *verify error*. Följande steg låter dig testa detta.

För att rensa BASIC-minnet, skriv `NEW` och tryck **Return**.

Därefter, skapa ett enkelt program.

```
10 PRINT "HEJ"
20 PRINT "HEJ DÅ"
```

Spara programmet till filen `"HEJ"`.

```
SAVE "HEJ",8
```

Kontrollera att innehållet i filen är samma som innehållet på disk.

```
VERIFY "HEJ",8
```

Din dator svarar med:

```
SEARCHING FOR HEJ
VERIFYING
OK
```

Modifiera programmet till att skriva ADJÖ istället för HEJ DÅ.

```
20 PRINT "ADJÖ"
```

Verifiera igen.

```
VERIFY "HEJ",8
```

Konstatera att innehållet på disk ni inte längre stämmer överens med vad som finns i minnet.

```
SEARCHING FOR HEJ
VERIFYING
?VERIFY  ERROR
```

Använd alltså kommandot VERIFY direkt efter SAVE om du vill vara säker på att programmet blev sparat korrekt.

Skapa och återkalla en datafil i BASIC

Oavsett om du ska läsa data från en fil eller skriva data till en fil, behöver du använda kommandot OPEN. Oavsett om du ska läsa eller skriva, måste du alltid avsluta med CLOSE. Kommandot PRINT# fungerar som PRINT, med en viktig skillnad:

PRINT skriver till den enhet som valts med CMD medan PRINT# skriver till en enhet som öppnats med OPEN. Enhetsnumret anges som första argument. Argumenten till PRINT# är först det logiska filnumret, därefter det data som ska skrivas.

Så för att kunna göra något, måste man veta hur OPEN fungerar, och det beskrivs i kapitlet om kommandon.

Slutligen, notera att PRINT# avslutar med ett radbryte, precis som PRINT. Detta är viktigt att veta för att kunna skriva korrekt kod för att återhämta data som skrivits till en fil. Detta exempel skapar en sekventiell fil med två poster. Den första posten innehåller texten HELLO WORLD och den andra posten innehåller texten GOODBYE CRUEL WORLD. Men låt oss först formatera en diskett. Infoga din diskett (som antingen är tom eller vars innehåll du kan avvara) i din diskdrive. I koden nedan antas enhetsnumret vara 8, och kom ihåg att all befintlig information på disketten kommer att försvinna:

```
OPEN 1,8,15,"N:LABB,00":CLOSE1
```

Därefter kan vi skapa en datafil med valfritt namn. I detta exempel antas enheten fortfarande vara 8 och namnet på filen är kort och gott TEST. Det tredje argumentet (sekundärt nummer) på rad 10 (kommandot OPEN), 2, är viktigt eftersom det anger att vi vill skriva till filen.

```
10 OPEN 1,8,2,"TEST"
20 PRINT# 1,"HELLO WORLD"
30 PRINT# 1,"GOODBYE CRUEL WORLD"
40 CLOSE 1
```

Om allt har gått som det ska, så har du nu en ny fil på din diskett. Filen heter TEST och innehåller texten som skrevs med PRINT# på rad 20 och 30. Om du kör programmet igen med andra PRINT#-satser kommer inget att hända –

programmet kan bara operera om filen vars filnamn anges i `OPEN`-satsen inte sedan tidigare.

Din dator vet inte om vad detta är för slags fil, och faktum är att den tror att filen är en programfil. Du kan fortfarande återkalla filen korrekt, med följande kod:

```
10 OPEN 1,8,8,"TEST"
20 INPUT# 1,A$:PRINT A$
30 INPUT# 1,A$:PRINT A$
40 CLOSE 1
```

Rad `10` öppnar filen för läsning. Denna gång anger vi `8` som sekundärt nummer (det tredje argumentet) för att indikera att vi vill läsa filen. Rad `20` och `30` är identiska, för vi vill att samma sak ska utföras två gånger. Först läser vi en textrad från filen och lagrar den i `A$` med hjälp av kommandot `INPUT#` (som alltså läser fram till nästa strängavgränsare) och sedan använder vi `PRINT` för att skriva ut resultatet på skärmen. Eftersom vi skrev två textsträngar till filen, måste vi läsa två textsträngar innan vi har återkallat all lagrad information. När du startat programmet med RUN kommer du se detta:

```
HELLO WORLD
GOODBYE CRUEL WORLD

READY.
```

Som vanligt är `READY` datorns sätt att indikera att den väntar på nya instruktioner.

Figur 32: Datorn anser felaktigt att datafilen är ett program.

Om du tittar på innehållet på disketten så listas filen `TEST` som om den vore ett program, vilket avslöjas av `PRG` till vänster om filen. Detta är inte något problem så länge du vet bättre, men det är bra om både användare och

system är överens om viken fil som innehåller vad. Den korrekta koden som skriver filen ser ut så här (fast nu heter filen i stället TEST2):

```
10 OPEN 1,8,2,"TEST2, SEQ, W"
20 PRINT# 1,"HELLO WORLD"
30 PRINT# 1,"GOODBYE CRUEL WORLD"
40 CLOSE 1
```

Förändringarna är gjorda på rad 10. Förutom att filnamnet är ändrat, så har strängen som utgör det fjärde argumentet kompletterats med filtyp (SEQ) och W (för skrivning). Nu vet diskdriven hur den ska märka upp den inkommande filen. *Observera att filnamn, filtyp och skrivning avgränsas med både ett kommatecken och ett blanksteg.* Om din diskdrive får TEST2,SEQ,W så kan den inte hantera det, men får den TEST2, SEQ, W så förstår den att den ska *skriva* till en *sekventiell* fil med namnet TEST2.

Motsvarande förändring i programmet som läser filen ser ut så här (icke ändrade rader är utelämnade):

```
10 OPEN 1,8,8,"TEST2, SEQ, R"
```

Denna information gör att diskdriven förstår att den ska *läsa* en *sekventiell* fil med namnet TEST2.

Om du läser en post med INPUT# efter att sista befintliga posten redan är läst, förändrar inte INPUT# värdet av variabeln. På det viset kan du kontrollera när all information i filen redan är läst. Det skulle kunna göras så här:

```
10 OPEN 1,8,8,"TEST2, SEQ, R"
20 A$="SLUT"
30 INPUT# 1,A$
40 IF A$="SLUT" THEN 70
50 PRINT A$
60 GOTO 20
70 CLOSE 1
```

Programmet läser allt innehåll i en sekventiell fil, oavsett hur mycket eller hur lite den innehåller. Förutom programfiler (PRG) och sekventiella filer (SEQ) finns även relativa filer (REL), användardefinierade filer (USR) och odokumenterade filer (DEL).

Skillnaderna mellan INPUT# och GET# är att INPUT# läser nästa post (typiskt, fram till nästa avgränsare) medan GET# läser nästa tecken (eller

byte), samt att `GET#` ger `199` som svar när filen är slut. Följande kod läser filen i fråga, tecken för tecken:

```
10 OPEN 1,8,8,"TEST2, SEQ, R"
30 GET# 1,A$
30 IF ASC(A$)=199 THEN 60
40 PRINT A$;
50 GOTO 20
60 CLOSE 1
```

Var noga med att använda rätt filtyper om du skapar något annat än en programfil, vilket är det du antas skapa om inget annat anges. Programfiler skiljer sig från andra filer genom att vara beroende till den minnesadress som programmet sparades ifrån, vilket också lagras i filens två första bytes.

För mer information, se avsnittet om `SAVE` i kapitlet om kommandon.

APPENDIX A: NYCKELORDSKODER

Appendix A: Nyckelordskoder

Bilaga A om nyckelordskoder vänder sig till dig som vill bygga egna verktyg som hanterar BASIC-filer, kanske på en Windows-maskin. Bilaga kan också vara intressant för avancerade Commodore-programmerare som vill arbeta med BASIC-filer på byte-nivå.

Ett BASIC-program lagras som text i minnet och på disk, undantaget de radnummer och nyckelord som förekommer i programmet. Radnummer lagras binärt och nyckelord ersätts med koder. Följande program innehåller radnumret `10`, nyckelordet `PRINT` samt uttrycket `A`:

```
10 PRINTA
```

Nyckelorden hittas även om de skrivits ihop med något annat. När programmet sparas på disk från en Commodore 64, är det följande bytes som skrivs ner:

Byte-värde (DEC)	Byte-värde (HEX)	Betydelse
1	01	Programmets startadress (låg byte)
8	08	Programmets startadress (hög byte)
8	08	Pekare till nästa programsats (låg byte)
8	08	Pekare till nästa programsats (hög byte)
10	0A	Radnummer (låg byte)
0	00	Radnummer (hög byte)
153	99	Kod för nyckelordet `PRINT`
65	41	Tecknet `A` enligt PETSCII

En nyckelordskod är ett tal mellan `128` och `203`. Alla nyckelordskoder är listade på nästa uppslag. I tabellen ovan är `PRINT` det enda nyckelordet som förekommer, som representeras av `153`. BASIC-programmets startadress (rad 1 och 2) ska på en Commodore 64 vara `2049`, såvida inte startadressen pekats om. Talet `2049` beskrivs av den höga byten `8` (8×256 = 2048) och den låga byten `1` (2048 + 1 = 2049), med den låga först. Pekaren till nästa programsats (rad 3 och 4) varierar med första programsatsens längd.

Pekaren som anger att BASIC-program börjar på adress `2049` ligger på en Commodore 64 på adress `43-44` (låg och hög byte) och pekaren som anger BASIC-areans slutadress finns på adress `55-56` och har värdet `40960` om det

inte ändrats. Man kan lagra BASIC-program till och med adress `40959` eftersom värdet (160×256 + 0) anger när man passerat tillåtet minne. På VIC-20 lagras BASIC-program normalt på adress `4097` till `7679`. Även på VIC-20 bestäms startadressen av värdet på adress `43-44`.

Tabellen över nyckelordskoder återges på nästa sida.

Skulle samma program (`10 PRINTA`) i stället ha sparats på en VIC-20, skulle filen innehålla följande bytes:

Byte-värde (DEC)	**Byte-värde (HEX)**	**Betydelse**
1	01	Programmets startadress (låg byte)
16	10	Programmets startadress (hög byte)
8	08	Pekare till nästa programsats (låg byte)
16	10	Pekare till nästa programsats (hög byte)
10	0A	Radnummer (låg byte)
0	00	Radnummer (hög byte)
153	99	Kod för nyckelordet `PRINT`
65	41	Tecknet `A` enligt PETSCII

Om inget annat har sagts, är alltså startadressen för ett VIC-20-program 4097 (16×256 + 1), upp till 7679 (30×256 + 0 ger 7680 men definierar när man passerat tillåtet minne).

HEX	DEC	Nyckelord
80	128	END
81	129	FOR
82	130	NEXT
83	131	DATA
84	132	INPUT#
85	133	INPUT
86	134	DIM
87	135	READ
88	136	LET
89	137	GOTO
8A	138	RUN
8B	139	IF
8C	140	RESTORE
8D	141	GOSUB
8E	142	RETURN
8F	143	REM
90	144	STOP
91	145	ON
92	146	WAIT
93	147	LOAD
94	148	SAVE
95	149	VERIFY
96	150	DEF
97	151	POKE
98	152	PRINT#
99	153	PRINT
9A	154	CONT
9B	155	LIST
9C	156	CLR
9D	157	CMD
9E	158	SYS
9F	159	OPEN
A0	160	CLOSE
A1	161	GET
A2	162	NEW
A3	163	TAB
A4	164	TO
A5	165	FN

HEX	DEX	Nyckelord
A6	166	SPC
A7	167	THEN
A8	168	NOT
A9	169	STEP
AA	170	+
AB	171	-
AC	172	*
AD	173	/
AE	174	^
AF	175	AND
B0	176	OR
B1	177	>
B2	178	=
B3	179	<
B4	180	SGN
B5	181	INT
B6	182	ABS
B7	183	USR
B8	184	FRE
B9	185	POS
BA	186	SQR
BB	187	RND
BC	188	LOG
BD	189	EXP
BE	190	COS
BF	191	SIN
C0	192	TAN
C1	193	ATN
C2	194	PEEK
C3	195	LEN
C4	196	STR$
C5	197	VAL
C6	198	ASC
C7	199	CHR$
C8	200	LEFT$
C9	201	RIGHT$
CA	202	MID$

APPENDIX B: MINNESADRESSER

Appendix B: Minnesadresser

Följande minnesadresser används i bokens exempel:

VIC-20

Exekveringsadress (pekare) för `USR`	1-2
Startadressen (pekare) för BASIC-program	43-44
Slutadressen (pekare) för BASIC-program	55-56
Påverkar innehållet i processorns register (A, X och Y)	780-782
Minnesbuffert för kassett-data (kan skrivas över)	828-1019
Textminnet	7680-8185
Startadressen (pekare) för teckenminnet	36869
Bakgrundsfärg och borderfärg	36879
Ljud-registret	36874-36878
Teckenfärgminnet	38400-38905
Funktion som flyttar textmarkören till positionen som är lagrad i processorns register	65520

Commodore 64

Startadressen (pekare) för BASIC-program	43-44
Slutadressen (pekare) för BASIC-program	55-56
Exekveringsadress (pekare) för `USR`	785-786
Påverkar innehållet i processorns register (A, X och Y)	780-782
Teckenminnet	1024-2023
Pekare till bilddata för sprites	2040-2047
Positioner på bildskärmen för sprites	53248-53264
Flaggor för att aktivera sprites	53269
Startadressen (pekare) för teckenminnet	53272
Bakgrundsfärg	53281
Borderfärg	53280
Val av skärmläge	53265
Ledigt minne för maskinkodsprogram	49152-53247
SID-registret (ljud)	54272-55295
Funktion som flyttar textmarkören till positionen som är lagrad i processorns register	65520

APPENDIX C: ORDFÖRKLARINGAR

Appendix C: Ordförklaringar

Adress

Den minsta allokeringsbara enheten i en 8-bitarsdator är en byte, varje byte har en *adress* (eller *minnesadress*). Vissa är läs- och skrivbara (RAM), andra är endast läsbara (ROM).

Användardefinierad variabel

En användardefinierad variabel är en variabel skapad av datoranvändaren. Här placeras värdet `23` i variabeln `AW`:

```
AW=23
```

En användardefinierad variabel är inte en systemvariabel.

Argument

Data som skickas till (till exempel) en funktion för att påverka dess funktionalitet. Ett argument är samma sak som en *parameter*.

Array

En array är en samling av variabler som identifieras av ett index. I BASIC har arrayen själv ett namn som följer reglerna för variabelnamn, där varje element identifieras av ett 0-baserat index som anges inom parentes, till exempel `A(31)`. Ordet *vektor* kan användas synonymt med array.

Binär logik

Binär logik är logik som är tvåställig i betydelsen att ett uttryck antingen är sant eller falskt, precis som en bit.

Binära talsystemet

Det binära talsystemet använder endast två tecken för att beskriva ett tal, till skillnad från det decimala talsystemet som använder tio eller det hexadecimala som använder sexton. Det innebär att de fem första talen (noll till fyra) skrivs 0, 1, 10, 11 och 100.

Bit

En bit är ett tal mellan 0 och 1. Åtta bitar utgör en byte. Det binära talsystemet utgörs av bitar, från engelskans "binary digits".

Bitvis

En bitvis operator agerar på värdets bitmönster, bestående av `0` eller `1`. och kan därför användas i binär aritmetik. Som exempel är `2 OR 4` lika med `6`, eftersom `00000010 OR 00000100` är lika med `00000110`. De binära bitvisa operatorerna är `AND` och `OR`, den unära är `NOT`.

Bitmaskning

Med bitmaskning avses metoder att läsa av individuella bitar i en byte, eller skriva till en eller flera bitar utan att påverka andra bitar. Bitmaskning måste behärskas av den som vill kunna sätta eller läsa av flaggor, som t.ex. vilken sprite som ska vara synlig.

Booleskt värde

Ett booleskt värde är ett värde som antingen är sant eller falskt. I Commodore BASIC 2.0 second release representeras 0 som falskt och icke-0 som sant vid test. Som testresultat är 0 falskt och -1 sant. Vid bit-operationer är 0 falskt och 1 sant.

Border

Ramen runt den yta som VIC-20 och Commodore 64 kan visa text och högupplöst grafik på benämns som *border*. Borderfärgen är ramens färg, medan bakgrundsfärgen avser färgen på ytan med text och grafik.

Byte

En byte är den minsta enheten med en egen adress. En byte består av åtta bitar (eller två nibbles), vilket innebär att en byte kan befinna sig i ett av 256 olika tillstånd, till exempel ett tal mellan 0 och 255.

Call stack

Varje hopp som görs med `GOSUB` lagras i datorns *call stack*, som är ett slags lista över vilka rader som anropats med `GOSUB`. När kommandot `RETURN` påträffas, plockas ett hopp från datorns call stack, och exekveringen fortsätter på raden efter. Både VIC-20 och Commodore 64 har avsatt 256 bytes för sin call stack, och klarar att hålla programhopp i 23 led i minnet.

Datasette

I detta sammanhang är ett datasette en speciell kassettbandspelare som kan användas för att lagra information eller program på band eller återställa information eller program från band.

Figur 33: Commodore 1530 C2N Datasette

Datatyp

Din Commodore-dator kan endast lagra ettor och nollor, vilket representerar talet `0` eller talet `1`. Dessa kallas bitar och är alltid grupperade om 8. Åtta bitar utgör en byte, och antalet tal som kan beskrivas med åtta bitar (en byte) är 256 (0-255). De individuella bitarna saknar minnesadress, men varje byte (åtta bitar) har var sin minnesadress. Så snart man vill läsa något annat än individuella tal mellan 0 och 255 (åtta bitar) måste man veta hur olika bytes ska kombineras. En datatyp är en konfiguration av kombinationer av bytes. till exempel består talet `16961` av en kombination av två bytes, nämligen `65` och `66` (66 * 256 + 65 = 16961), medan textsträngen `AB` också består av samma kombination av bytes. Om man vet vilken datatyp man hanterar, kan man avkoda informationen korrekt.

Diskdrive

En diskdrive (eller en *diskettstation*) är det bästa *sekundärminnet* för VIC-20 och Commodore 64. Den mest populära modellen hette Commodore 1541 och fungerade tillsammans med 5¼"-disketter som rymde knappt 165 kilobyte per sida.

Figur 34: En diskdrive av modell Commodore 1541. Foto: Wojciech Pędzich

Figur 35: Floppydisk på 5¼" som passar till Commodore 1541. Foto: Andreas Frank

En floppydisk är det skivminne som stöds av Commodore, och den kan ibland benämnas *disk* eller *diskett*.

DOS

DOS står för *Disk Operating System* och med DOS avser operativsystem kapabla att serva användaren i användandet av skivminnen som sekundärminne. Commodore-maskinerna erbjuder BASIC-tolken som användarens gränssnitt för att använda skivminnen (floppydisk), och den diskdrive som är inkopplad till enheten, erbjuder ytterligare kommandon som kan skickas till enheten för utförande. Exakt vilka kommandon som står till förfogande beror på vilken modell av diskdrive som är inkopplad till datorn. Några modeller som kan förekomma är 1540, 1541 som normalt såldes till VIC-20 och Commodore 64, 1541 II, den externa 1571 och den interna 1571 som normalt såldes till Commodore 128 respektive 128D. Denna bok tar avstamp i modell 1541.

Enhetscirkeln

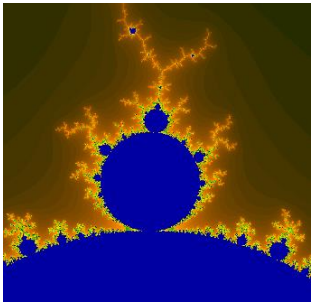
Inom geometri är enhetscirkeln en cirkel med radie 1. För att beräkna en koordinat längs med cirkeln kan man använda sig av cosinus och sinus, som i Commodore BASIC 2.0 second release är implementerade i funktionerna `COS` och `SIN`.

Fibonaccisekvens

En fibonaccisekvens är en serie av tal som börjar med 0 och 1. Nästa tal är summan av de två föregående. Eftersom 0 + 1 är 1 blir nästa tal 1. Nästa tal i sekvensen blir 2 eftersom 1 + 1 är 2. De tjugo första fibonaccitalen är 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584 och 4181.

Fraktal

En fraktal är ett mönster som liknar sig själv i alla skalor, och som skapas genom programkod (till exempel BASIC) som itereras.



Figur 36: Mandelbrotsfraktalen.

Det finns naturliga strukturer som är fraktala, till exempel romanesco (nästa sida).

Figur 37: Romanesco är ett kolhuvud med fraktala egenskaper.

Fysisk fil

En *fysisk fil* är, till skillnad från en *logisk fil*, en faktisk fil som finns lagrad på floppydisk eller på kassettband. En fysisk fil har ett namn och en startadress eller en (förhoppningsvis) specifik struktur.

Grafikläge

En Commodore-maskin har olika grafiklägen som anger vad som kan visas på skärmen. I textläget (som används vid uppstart) kan VIC-20 visa text i 22 rader med 23 kolumner eller 25 rader och 40 kolumner för Commodore 64. I bitmapsläge hanterar man i stället individuella pixlar i två olika upplösningar, där flerfärgsläget har hälften så många individuella pixlar på skärmen som är möjligt i enfärgsläget (eller det högupplösta läget).

- Commodore 64, enfärgsläget: 320×200 pixlar
- Commodore 64, flerfärgsläget: 160×200 pixlar
- VIC-20, enfärgsläget: 176×184 pixlar
- VIC-20, flerfärgsläget: 88×184 pixlar

Hard reset

En *hard reset* innebär att datorn återställs och att hela datorns minne (BASIC-program och övrig data) raderas. Jämför med *soft reset*.

Hexadecimala talsystemet

Det hexadecimala talsystemet använder hela sexton olika tecken för att beskriva ett tal, till skillnad från det decimala talsystemet som använder tio eller det binära talsystemet som använder två. Det innebär att de tjugo första talen (noll till nitton) skrivs 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12 och 13.

Interrupten

Interrupten är en funktion för multitasking som finns inbyggd i VIC-20 och Commodore 64. Huvudprocessorn kan, med jämna mellanrum, släppa in andra uppgifter, vilket liknar det som idag kallas *time slicing*.

I/O

I/O är en förkortning av input/output och avser operationer som läser eller skriver från/till externa enheter som till exempel tangentbord, printer, skärm eller floppydisk.

Jiffy

Vad en *jiffy* (flera *jiffies*) är varierar från plattform till plattform, men i Commodore BASIC 2.0 second release är en jiffy en sextionsdels sekund. En jiffy är alltså 1/60 sekunder och 60 jiffies är en sekund. En mer samtida godtycklig tidsenhet är en *tick*, som ofta är kortare, till exempel en tusendels sekund.

Kilobyte

En *kilobyte* (kort *K*) är 1024 bytes. 16 K betyder således 16 kilobytes eller 16384 bytes.

Konsol

Konsolen (textkonsolen) är gränssnittet mellan användaren och datorn som användaren använder genom datorns tangentbord. Man känner igen konsolen på den blinkande markören.

Loader

En *loader* är ett program som laddar in ett program i minnet. I Commodore-världen är detta typiska exempel:

- Ett dataspel som ska laddas från kassett. Första programmet har till uppgift att visa en pausbild på skärmen, medan det hämtar in huvudprogrammet från kassettbandet.
- Ett BASIC-program som skriver ett maskinkodsprogram till minnet, som visas i avsnittet om `DATA`.

Logisk fil

Normalt är en fil ett stycke namngivet data på floppydisk, men en *logisk fil* kan vara precis vad som helst som kan läsas från eller skrivas till *som om* det vore en fil på en floppydisk. Förutom olika typer av filer, kan en logisk fil vara en kanal till en skrivare.

Markör

Textmarkören anger var nästa tecken från tangentbordet kommer att hamna. Markören illustreras som en blinkande rektangel.

Mikroprocessor

En mikroprocessor är datorns centralenhet (Central Processing Unit eller CPU) som läser och utför instruktioner.

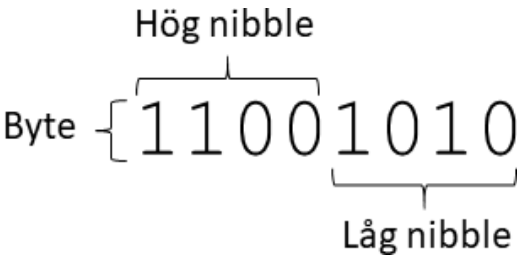
Figur 38: Mikroprocessorn MOS 6502 som återfinns i bland andra Atari 2600, VIC-20 och Apple II.

Multitasking

Multitasking innebär att två program körs samtidigt. Det kan till exempel vara ett maskinkodsprogram som körs samtidigt som ett BASIC-program. När man programmerar en Commodore-maskin pratar man om *interrupten*, som lite slarvigt är en svensk bestämd form av engelskans *interrupt*, vilket i sammanhanget närmast kan liknas med *att bryta in*. I enkelhet kan man tänka att datorn är utrustad timers som med jämna mellanrum kan exekvera din maskinkod. Det innebär att flera maskinkodsrutiner, och även någon BASIC-rutin, kan exekvera simultant. Detta sker på bekostnad av prestandan.

Nibble

En nibble består av fyra bitar. En byte består av två nibbles.



Figur 39: En byte består av två nibbles.

NTSC

NTSC, National Television Systems Committee, är en standard för analog färg-tv som primärt används i Amerika. NTSC har lite högre uppdateringsfrekvens och lite lägre upplösning än PAL som främst används i Europa. I Asien används SECAM.

Operand

En *operand* är ett värde i en ekvation. I följande exempel är `1` och `2` operander, medan `+` är *operator*:

```
X = 1 + 2
```

Operator

En *operator* är symbol eller en funktion i en matematisk operation. I följande exempel är `+` operator medan `1` och `2` är *operander*:

```
X = 1 + 2
```

PAL

PAL, Phase Alternate Line, är en standard för analog färg-tv som primärt används i Europa. I Amerika används NTSC och i Asien används SECAM.

Parameter

Data som skickas till (till exempel) en funktion för att påverka dess funktionalitet. En parameter är samma sak som ett *argument*.

Pekare

Pekare är inget som BASIC har stöd för, men man kommer ofta i kontakt med pekare när man använder datorns inbyggda funktioner. En minnesadress som syftar till att hålla reda på en minnesadress är en pekare. Olika pekare används på olika sätt. Ett par exempel som specifikt gäller Commodore 64: Adress `785`-`786` ett 16-bitarstal som är den exakta adressen för funktionen USR, medan på adress `2047` talar om var i minnet bilddata för den åttonde spriten på ligger, om man multiplicerar värdet med 64.

PETSCII

`PETSCII` (PET Standard Code of Information Interchange) är namnet på teckentabellen som används av bland andra VIC-20 och Commodore 64. PETSCII är Commodores version av ASCII, och teckentabellen är uppkallad efter den första datorn som använde den, Commodore PET (år 1977).

Pixel
Bilden som visas på skärmen består av punkter av olika färger, i en matris. Varje punkt (egentligen *bildelement*) som utgör bilden, kallas för *pixel*.

Primärminne
Primärminnet är datorns arbetsminne (datorns RAM-minne).

Pseudografik
Pseudografik är textbaserad grafik som syftar till att se ut som högupplöst rastergrafik. På din Commodore-dator finns massor av tecken framtagna för att kunna passa till pseudografik. Se kapitlet om text för mer information.

RAM
RAM står för *random access memory* och ordet avser minne som man kan skriva till eller läsa av i valfri ordning – man kan ange värdets minnesadress. I boken menas *datorns primärminne* eller *arbetsminne*.

Reset
En *reset* återställer datorn, vilket innebär att BASIC-program går förlorat. Se *hard reset* och *soft reset*.

ROM
ROM står för *read only memory* (minne som endast kan läsas, inte skrivas) och avser datorminne som beskriver datorns inbyggda funktioner.

Sekundärminne
Sekundärminnet används, till skillnad från primärminnet, till långvarig lagring. Commodore använder antingen floppydisk eller datasette som sekundärminne.

Sekventiell fil
Definitionsmässigt utgörs en sekventiell datafil innehåll av data i den ordningen den skrevs till filen. En konsekvens av detta är att filen måste läsas samma ordning som den skrevs. Detta är utmärkande för programfiler, textfiler och till exempel bildfiler.

Soft reset
En *soft reset* innebär att datorn återställs och att datorns BASIC-minne raderas. Jämför med *hard reset*.

Sprite

En sprite är en liten grafisk symbol som kan röra sig fritt på skärmen, utan att störa annan grafik. Commodore 64 kan visa åtta sprites men VIC-20 saknar sprites – vill man ha tillgång till sprites på VIC-20 så får man bygga en egen sprite-rutin. Se appendix E, som jämför VIC-20, Commodore 64 och Commodore 128 med varandra för mer information.

Systemvariabel

En systemvariabel är en variabel som får sitt värde från systemet snarare än från datoranvändaren. En systemvariabel är inte en *användardefinierad* variabel. Vissa systemvariabler kan initieras eller uppdateras av användaren, andra systemvariabler är helt skrivskyddade, men alla uppdateras av systemet.

Vektor

I BASIC används ordet *vektor* synonymt med ordet *array*. En array (eller en vektor) har ett antal element och varje element har ett värde.

APPENDIX D: COMMODORE BASIC 7.0

Appendix D: Commodore BASIC 7.0

Datorn Commodore 128 introducerades på marknaden år 1985 och är en vidareutveckling av Commodore 64. När det gäller multimediakapacitet är de två datorerna snarlika, men några förbättringar som sticker ut är den dubbla minneskapaciteten, det stora tangentbordet, 80-kolumnsläget, den extra Z80-processorn, en utökad BASIC och att den kan köras i flera olika lägen:

- Commodore 128-läget ger tillgång till en utökad BASIC, Commodore BASIC 7.0 som bland annat erbjuder kommandon för fil- och diskhantering och multimedia. Commodore 128-läget kan användas med både 40 kolumner text och 80 kolumner text. I 80-kolumnsläget erbjuds inte (utan vidare) bitmapsgrafik.
- Commodore 64-läget erbjuder möjligheten att köra program utvecklade för Commodore 64, av vilka det finns massor att välja på.
- CP/M-läget (möjligt tack vare Z80-processorn) erbjuder avancerade diskoperationer och möjligheten att köra CP/M-program som till exempel Turbo Pascal och Microsoft BASIC.

Commodore 128 levereras med en nyare version av Commodore BASIC, Commodore BASIC 7.0, som både innehåller nya nyckelord och nyckelord från version 4. Dessa 105 nyckelord finns i version 7.0, men inte i version 2.0 second release (1980):

`APPEND`	*Kompletterar innehållet i en fil.*
`AUTO`	*För automatisk radnumrering.*
`BACKUP`	*Kopierar en disk.*
`BANK`	*Kommando för minneshantering på datorer med utökat minne.*
`BEGIN`	*BEGIN används tillsammans med BEND för att gruppera programsatser som ska köras efter en IF-sats.*
`BEND`	*BEND stänger ett block som öppnats med BEGIN.*
`BLOAD`	*Läser in en binär fil till en angiven plats i minnet. Används till exempel för att läsa in maskinkodsprogram eller media.*
`BOOT`	*Laddar in ett program flaggat som "bootbart" från disk. Om en diskdrive är inkopplad försöker Commodore 128 att "boota" vid uppstart.*
`BOX`	*Ritar en rektangel på skärmen.*
`BSAVE`	*Sparar innehållet i angivna adresser i RAM till disk. Används till exempel för att spara maskinkodsprogram eller media.*
`BUMP`	*Testar om två sprites har kolliderat med varandra.*
`CATALOG`	*Listar innehållet på en diskdrive utan att tömma BASIC-minnet. Är synonymt med DIRECTORY.*

CHAR	*Ritar ett tecken på skärmen*
CIRCLE	*Ritar en cirkel på skärmen.*
COLLECT	*Frigör utrymme på en disk som korrupta filer tar upp.*
COLLISION	*Identifierar sprite-kollisioner eller kollision med ljuspenna och hoppar till önskad programrad om så inträffat.*
COLOR	*Väljer färg ur färgpaletten att använda som border, bakgrund, grafikfärg eller textfärg. Commodore 128 delar färgpalett med Commodore 64 och VIC-20 när Commodore 128 används i 40-kolumnsläge. I 80-kolumnsläge (som endast kan visa text) har Commodore 128 en annorlunda färgpalett.*
CONCAT	*Slår samman två datafiler.*
COPY	*Kommandot COPY används på system med två diskettstationer för att kopiera en fil från den ena disketten till den andra.*
DCLEAR	*Rensar alla öppna filer på angiven drive. Man kan även specificera enhetsnummer.*
DCLOSE	*DCLOSE stänger en fil som varit öppen för läsning eller skrivning.*
DEC	*Tar en sträng innehållande ett hexadecimalt tal och ger ett decimalt heltal av motsvarande värde. DEC("A") ger 10. Den funktion som konverterar tillbaka till hexadecimal heter HEX$.*
DELETE	*Kommandot raderar en eller flera rader av BASIC-kod från minnet. Man kan ange ett specifikt radnummer eller ett omfång.*
DIRECTORY	*Listar innehållet på en diskdrive utan att tömma BASIC-minnet. Är synonymt med CATALOG.*
DLOAD	*Hämtar ett namngivet BASIC-program från disk (med möjlighet att specificera enhetsnummer och disknummer (i händelse av att enheten är dubbelsidig, som till exempel Commodore 1571). Motsvarar LOAD "filnamn",8,0 på VIC-20/C64.*
DO	*Startar antingen en evighetsiteration, en 0-till-flera-iteration eller en 1-till-flera-iteration, som stängs med LOOP. Tillsammans med WHILE eller UNTIL är iterationen av 0-till-flera-typ, och ensamt är den antingen en evighetsiteration eller en 1-till-flera-iteration, beroende på vad som följer efter LOOP, om något.*
DOPEN	*Öppnar en logisk fil av valfri typ (sekventiell, program, o.s.v.) för läsning och/eller skrivning.*
DRAW	*Ritar en pixel, en linje eller en polygon med valfri färg ur färgpaletten.*
DS	*Presenterar det senaste DOS-felets nummer.*
DS$	*Presenterar en beskrivande text om det senaste DOS-felet.*
DSAVE	*Sparar ett BASIC-program till angiven enhet.*
DVERIFY	*Jämför BASIC-programmet i minnet med ett BASIC-program i en fil, för att säkerställa att programmet är korrekt lagrat.*
EL	*Ger radnummer för senast inträffade fel.*
ELSE	*ELSE används tillsammans med IF om man vill utföra ett kommando om villkoret efter IF är falskt.*
ENVELOPE	*Definierar karaktären för ett musikinstruments volym.*
ER	*Ger felkod för senast inträffade fel.*
ERR$	*ERR$ är ren funktion som ger det felmeddelande vars kod man skickar in som argument. Tillsammans med ER ger ERR$ en beskrivning av det senast inträffade felet.*
EXIT	*Används mellan DO och LOOP. Kommandot lämnar en pågående iteration som öppnats med DO och fortsätter exekveringen efter LOOP.*
FAST	*FAST saknades av någon anledning i Commodore BASIC 2.0 second release trots att det var ett känt inslag för 1980-talets BASIC-*

	programmerare, inte minst på Sinclair. FAST stänger av bilduppdateringen och sätter processorn i 2MHz-läge, så att BASIC-programmet exekverar snabbare, till priset av att skärmen är helt blank till dess at kommandot SLOW användes.
FETCH	*För expanderade maskiner: Flyttar innehåll från utökat RAM till datorns RAM.*
FILTER	*Konfigurerar ljudchipets (SID) filterparametrar. Filter tar parametrar som styr frekvens, filtertyp och resonans.*
GETKEY	*Tar en lista med variabler som argument, och sparar en tangenttryckning i varje variabel.*
GO64	*Kommandot växlar till Commodore 64-läge, så att mjukvara för Commodore 64 kan köras. Se avsnitt nedan.*
GRAPHIC	*Väljer grafikläge (textläge, bitmapsläge, med flera).*
GSHAPE	*Hämtar grafikdata från en strängvariabel (som sparats med SSHAPE) och ritar ut den på skärmen.*
HEADER	*Formaterar en floppydisk.*
HELP	*Om ett program bryts på grund av programkörningsfel, visar HELP den felaktiga raden.*
HEX$	*Tar ett heltal och ger en sträng innehållande ett hexadecimalt tal av motsvarande värde. HEX$(10) ger A. Den funktion som konverterar tillbaka till decimal heter DEC.*
INSTR	*Ger en strängs position i en annan sträng.*
JOY	*Ger aktuell riktning för valfri joystick.*
KEY	*Visar eller kopplar kommandon till funktionstangenterna (F1 till F8).*
LOCATE	*Placerar pixelmarkören på angiven plats, som kan användas för en ritoperation (till exempel DRAW).*
LOOP	*Stänger antingen en evighetsiteration, en 0-till-flera-iteration eller en 1-till-flera-iteration, som öppnats med DO. Tillsammans med WHILE eller UNTIL är iterationen av 1-till-flera-typ, och ensamt är den antingen en evighetsiteration eller en 0-till-flera-iteration, beroende på vad som följer efter DO, om något.*
MONITOR	*Startar den inbyggda maskinkodsmonitorn.*
MOVSPR	*Positionerar eller flyttar en sprite på skärmen.*
PAINT	*Fyller en area, avgränsad av pixlar, med en färg.*
PEN	*Ger X- och Y-positionen för ljuspennan (som är ett pekdon för CRT-skärmar).*
PLAY	*Spelar upp musik som beskrivs av en textsträng.*
POINTER	*Ger minnesadressen för en angiven variabel.*
POT	*Läser av paddelns position. (En paddel är en ratt som används för analog styrning i två riktningar, till skillnad från en styrspak som används för digital styrning i fyra riktningar.)*
PRINT USING	*Skriver ut formaterad text.*
PUDEF	*Anger val av symboler och skiljetecken för PRINT USING.*
RCLR	*Läser av aktuell färgpalett för olika grafiklägen.*
RDOT	*Läser grafikmarkörens position och underliggande färg.*
RECORD	*Placerar filpekaren när du arbetar med en öppen sekventiell fil.*
RENAME	*Ändrar namnet på en fil på disk.*
RENUMBER	*Används i direktläge för att numrera om programsatserna på önskat vis.*

RESUME	*Används för felhantering i runtime-läge. Kommandot hoppar till en önskad rad efter att ett fångat fel (TRAP) är hanterat.*
RGR	*Håller aktuellt skärmläge (0-5).*
RREG	*Läser innehållet i processorns ackumulator, statusflaggor och register.*
RSPCOLOR	*För en sprite i flerfärgsläge ger RSPCOLOR information om färg 1 eller 2.*
RSPPOS	*Ger information om position och hastighet för valfri sprite.*
RSPRITE	*Ger information om inställningar för valfri sprite.*
RWINDOW	*Ger information om aktuellt textfönster.*
SCALE	*Slår på och/eller ställer in grafikskalning, eller stänger av grafikskalning.*
SCNCLR	*Rensar skärmen från text (eller grafik, om så önskas).*
SCRATCH	*Raderar en fil från en diskett.*
SLEEP	*Pausar BASIC-programmet i ett önskat antal sekunder.*
SLOW	*SLOW återaktiverar bilduppdateringen och sätter tillbaka processorns hastighet till 1MHz efter att kommandot FAST har använts.*
SOUND	*Spelar en ljudeffekt.*
SPRCOLOR	*Anger vilken färg/vilka färger en sprite ska visas med.*
SPRDEF	*Startar Commodore 128:s inbyggda sprite-editor.*
SPRITE	*Slår på och eller konfigurerar en sprite, eller slår av en sprite.*
SPRSAV	*Kopierar pixeldata i en strängvariabel till en av åtta tillgängliga sprites.*
SSHAPE	*Sparar pixeldata i en strängvariabel, som sedan kan ritas ut med GSHAPE.*
STASH	*För expanderade maskiner: Flyttar innehåll från datorns RAM till utökat RAM.*
SWAP	*För expanderade maskiner: Ersätter innehåll i datorns RAM med innehållet i inkopplat utökat RAM.*
TEMPO	*Anger hur snabbt musik spelas av PLAY-kommandot. 0 är långsammast och 255 är snabbast.*
TRAP	*TRAP, TRON och TROFF är kommandon för felsökning (debugging). Kommandot TRAP hoppar till ett angivet radnummer i händelse av att ett fel uppstår.*
TROFF	*TROFF inaktiverar TRON.*
TRON	*TRON aktiverar felsökningsläge, vilket innebär att Commodore 128 för varje rad som exekveras, skriver ut radnumret för raden i fråga. Felsökningsläget inaktiveras med TROFF.*
VOL	*Anger ljudstyrkan för nästa ljud som spelas. VOL påverkar SOUND och PLAY.*
WHILE	*En iteration som exekverar så länge ett binärt uttryck utvärderas som sant. WHILE avslutas med WEND.*
WIDTH	*Anger horisontell bredd på vektorgrafik (som till exempel LINE eller CIRCLE). Kan vara 1 (enkel bredd) eller 2 (dubbel bredd).*
XOR	*En logisk binär operator som ger sant som svar om operanderna är olika. Falskt xor falskt är falskt, sant xor sant är falskt, sant xor falskt är sant och falskt xor sant är sant.*

Kommandot `GO64` är lite speciellt och dessutom är kommandot `SYS` utökat med flera parametrar. Dessa utvecklas lite längre fram i detta kapitel.

Med dessa nya kommandon och funktioner på plats, öppnas oanade möjligheter. *Lorenz-attraktorn* är en förenklad modell som beskriver typ konvektionsströmmar i till exempel luft eller vatten, som kan implementeras som ett enkelt program. Med kapaciteten i Commodore BASIC 7.0, kan detta formuleras så här (fritt översatt från Bo E. Carlsson[30])[31]:

```
10 GRAPHIC 1,1
20 X=5
30 Y=5
40 Z=5
50 T=0
60 S=1/200
70 D=10
80 R=28
90 B=8/3
100 T=T+0.1
110 DX=D*(Y-X)
120 X1=X+DX*S
130 DY=(R*X-Y)-X*Z
140 Y1=Y+DY*S
150 DZ=X*Y-B*Z
160 Z1=Z+DZ*S
170 X=X1
180 Y=Y1
190 Z=Z1
200 DRAW 1,150+4*X,20+3*Z
210 IF T<1000 GOTO 100
```

Rad 10 väljer grafikläget bitmapsläge (högupplöst enfärgsläge).

Rad 20-90 initierar variablerna vi behöver under körning.

Rad 100 inleder iterationen och därefter appliceras attraktorn.

Rad 200 ritar en punkt på skärmen.

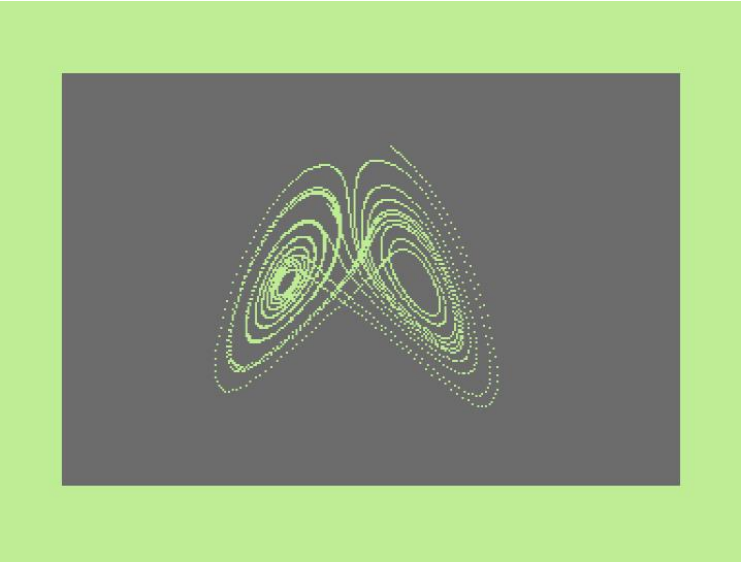
Rad 210 upprepar, givet att variabeln T är mindre än 1000. Eftersom T ökar med en tiondel per iteration, kommer iterationen att köras 10 000 gånger.

Programmet tar en stund att exekvera, men du ser en figur byggas upp på skärmen medan programmet körs. Ovanstående kod ger efter en stund resultatet som visas på nästa sida, och figuren fortsätter att fyllas i tills tiotusen bildpunkter (pixlar) är ritade på skärmen eller tills programmet avbryts.

En okommenterad version av detta program för Commodore 64 finns i kapitlet om grafik i avsnittet om grafik på Commodore 64.

[30] "Fraktaler och bilder av kaos och kosmos" av Bo E. Carlsson 1992, ISBN 91-518-2491-4.

[31] https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/lorenz.c128.bas



Figur 40: Lorenz-attraktorn, implementerad på Commodore 128.

För att avbryta körningen i förtid, eller för att återgå till textläge efter körningen, håll ner **Run Stop** och tryck **Restore**.

GO64

`GO64` är ett kommando i Commodore BASIC 7.0 som sätter Commodore 128 i Commodore 64-läge. Precis som `GOTO` även kan skrivas som `GO TO` så kan `GO64` även skrivas som `GO 64`. Kommandot kan användas både i direktläge och i runtime-läge. Om `GO64` använd i direktläge ställer datorn en kontrollfråga innan den växlar till Commodore 64-läget, vilket är bra eftersom det BASIC-program du eventuellt har i minnet går förlorat när du växlar läge. Men **tänk på att** `GO64` **inte ger denna varning när det används i runtime-läget**, så spara ditt program innan du provkör det!

På grund av hur kommandotolken fungerar i Commodore BASIC är den avslutande delen, 64, nästan att betrakta som ett argument. Du kan växla till Commodore 64-läge genom att t.ex. skriva så här:

```
A=64:GOA
```

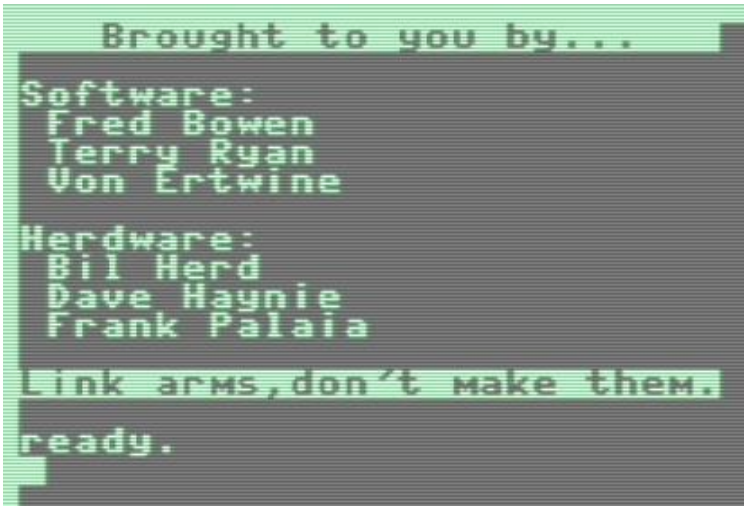
Eller så här:

```
GO8*8
```

Men om du räknar fel, uppstår felet *syntax*.

SYS

Kommandot `SYS` gör samma sak i Commodore BASIC 7.0 som i 2.0, men kommandot är utökat. Medan `SYS` på en Commodore 64 endast kan exekvera ett maskinkodsprogram på den angivna adressen (t.ex. `SYS 4096`) så kan `SYS` hantera parametrar till programmet som startas från Commodore BASIC 7.0. Detta innefattar att skriva till processorns register. Commodore 128 har ett inbyggt påskägg[32] som utnyttjar detta och kan aktiveras med `SYS 32800,123,45,6`. Påskägget visar en anspråkslös uppmaning samt namnen på dem som designat maskinen. Ordet *hardware* (hårdvara) är medvetet felstavat.



Figur 41: Ett dolt budskap i din Commodore 128.

[32] Ett påskägg är ett dolt meddelande i t.ex. en mjukvara eller en film.

APPENDIX E: EN JÄMFÖRELSE MELLAN VIC-20, COMMODORE 64 OCH COMMODORE 128

Appendix E: En jämförelse mellan VIC-20, Commodore 64 och Commodore 128

Tabellen nedan visar specifikationerna för VIC-20 (V20), Commodore 64 (C64) och Commodore 128 (C128).

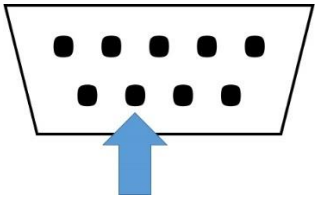
	VIC-20	C64	C128
Lanseringsår	1980	1982	1985
Programmeringsspråk	Commodore BASIC 2.0 second release	Commodore BASIC 2.0 second release	Commodore BASIC 7.0
Längsta programsats	88 tecken (4 rader)	80 tecken (2 rader)	160 tecken (4 rader i 40-kolumnsläge, 2 rader i 80-kolumnsläge)
Operativsystem	-	GEOS (tillval)	GEOS (tillval), CP/M 3.0
Huvudprocessor	MOS 6502	MOS 6510	MOS 8502, Z80B
Klockfrekvens[33]	1,1 MHz	0,99 MHz	1-2 MHz, 4 MHz
ROM	20 KB	20 KB	72 KB
RAM	5 KB	64 KB	128 KB
Expansionsmöjlighet RAM	32 KB	320 KB	512 KB
Text	22×23	40×25	40×25, 80×25
Skärmupplösning	176×184 plus border	320×200 plus border	320×200 plus border, 640×200
Monokrom grafik	176×184	320×200	320×200
Flerfärgsgrafik	88×184	160×200	160×200
Videoutgång	Analog (A/V)	Analog (RF, A/V)	Analog (RF, A/V)/Digital (RGBI)
Ljud	Tre fyrkantsvågor och ett brusljud, tre kanaler	Konfigurerbar fyrkantsvåg, triangelvåg, sinusvåg, brus, filter, med mera. Tre kanaler	Konfigurerbar fyrkantsvåg, triangelvåg, sinusvåg, brus, filter, med mera. Tre kanaler
Maskinkodsmonitor	-	-	Ja
Sprites	-	8	8
Sprite-editor	-	-	Ja

[33] Processorns hastighet är aningen olika för PAL-anpassade datorer och NTSC-anpassade datorer. För Commodore 128 gäller 4 MHz när Z80B-processorn används.

Följande anslutningsmöjligheter och knappar finns på de respektive maskinerna:

	VIC-20	C64	C128
Styrport 1 (styrspak, mus, ljuspenna)	Ja	Ja	Ja
Styrport 2 (styrspak, mus, ljuspenna)	Nej	Ja	Ja
Reset-knap	Nej	Nej	Ja
Powerknapp	Ja	Ja	Ja
Expansionsport för cartridges	Ja	Ja	Ja
Dataset	Ja	Ja	Ja
Seriebuss (för t.ex. diskdrive)	Ja	Ja	Ja
Ljud- och videoport	Ja (egen pin-layout)	Ja	Ja
RF-kontakt för tv	Nej	Ja	Ja
RGBI	Nej	Nej	Ja
Userport (för t.ex. modem)	Ja	Ja	Ja

Den digitala videoutgången på Commodore 128, RGBI, liknar till utseendet CGA-utgången på en IBM PC, men kabeln passar inte. RGBI-kontakten ger en digital signal men den har också en kompositsignal, vilket innebär att du kan få ut en svartvit högupplöst 80-kolumnssignal till en vanlig tv. Bilden (nästa sida) visar RGBI-kontakten på en Commodore 128.



Figur 42: Den sjunde pinnen innehåller en kompositsignal som kan tas emot av en tv.

Samtliga maskiner kan visa totalt 16 färger, med vissa begränsningar, enligt följande lista:

- **VIC-20** kan visa en av 16 färger som förgrund och bakgrund och en av 8 färger som border.
- **Commodore 64** kan visa en av 16 färger som förgrund, som bakgrund och som border.
- För **Commodore 128** i **40-kolumnsläge** gäller samma som för Commodore 64.
- **Commodore 128** i **80-kolumnsläge** saknar border och har alltid svart bakgrundsfärg. Text kan visas i färg från följande tabell:

Färgkod	**Motsvarande färg**
0	Svart
1	Vit
2	Mörkröd
3	Ljusturkos
4	Ljuslila
5	Mörkgrön
6	Mörkblå
7	Ljusgul
8	Mörklila
9	Brun
10	Ljusröd
11	Mörkturkos
12	Grå
13	Ljusgrön
14	Ljusblå
15	Ljusgrå

Commodore 128 är endast för att endast visa *text* i 80-kolumnsläge, inte vektorgrafik eller sprites, och för att färgerna ska återges krävs en monitor med RGBI-kontakt, till exempel Commodore 1084S. Med det sagt är hårdvaran kapabel att visa högupplöst grafik, men det finns inga BASIC-kommandon som drar nytta av detta.

Ämnet färger och grafik på Commodore 128 omfattas inte av den här boken, men tillgängliga BASIC-kommandon är listade i appendix D och boken *Commodore 128 Programmer's Guide* (ISBN 0-87455-031-9) tar upp det som är unikt för just Commodore 128, bland annat beträffande grafik.

APPENDIX F: MASKINKOD

Appendix F: Maskinkod

Maskinkod på en Commodore-maskin är 8-bitars talserier i RAM (alltså datorns arbetsminne) som representerar operationer och eventuellt en tillhörande parameter. Det innebär att varje instruktion alltid tar upp mellan 1 och 3 bytes:

- Operationskoden tar upp en byte
- Eventuell parameter tar upp en eller två bytes

En programsats är alltså en operationskod som anger vad processorns ska göra, och eventuellt ett argument till operationskoden. Argumentet, om något sådant behövs, är antingen ett 8-bitarstal eller ett 16-bitarstal (se särskilt kapitel).

Programmet exekveras av processorn, MOS Technology 6502.

Det är ganska vanligt att operationer har olika operationskoder, eftersom operationerna kan användas i olika lägen, så operationskoden anger inte bara vilken operation som avses, utan även vilket läge som avses.

För att skriva ett program behöver du veta var det finns ledigt minne. Både VIC-20 och Commodore 64 levereras med manual som innehåller en minneskarta. På VIC-20 ligger en buffert för kassett-data på adress 828-1019 som kan användas och på Commodore 64 är adress bland annat 49152-53247 tillgängligt för egna maskinkodsprogram.

Ett datorprogram bygger väldigt mycket på vilka funktioner man har byggt in i datorn, som till exempel grafik och ljud. På en Commodore-maskin kommer man typiskt åt dessa genom att sätta bytes på olika adresser i minnet.

Processorn själv har ett antal resurser till sitt förfogande, som operationskoderna använder. Det finns tre register (en byte vardera) som heter `A`, `X` och `Y`, samt ett antal specialregister som till exempel innehåller olika statusflaggor.

Följande resonemang kan följas oavsett vilken Commodore-maskin du äger, men exemplen fungerar endast på en Commodore 64.

Om vi vill skriva ett program på adress 49152 som lagrar värdet 7 i en specifik minnesadress, 49200, skulle det maskinkodsprogrammet kunna se ut så här:

```
169 7
141 48 192
96
```

Första programsatsen lagrar talet 7 i register `A` (ackumulatorn). 169 är operationskoden *Load Accumulator* (`LDA`) och 7 är argumentet.

Andra programsatsen lagrar innehållet i register `A` (som nu är 7) på adress 49200. 141 är operationskoden *Store Accumulator* (`STA`) och argumenten är 48 och 192. Anledningen till att 8-bitarstalen 48 och 192 motsvarar 16-bitarstalet 49200 är att den höga byten (192) är värd sig själv gånger 256 (49152) och den låga byten (48) är värd sig själv. 49152 + 48 = 49200.

Den tredje och sista programsatsen avslutar programmet. 96 är operationskoden för *Return From Subroutine* (`RTS`), som också avslutar programmet om programmet inte är inne i någon subrutin. `RTS` finns inte i olika lägen, utan har alltid operationskod `96`.

Vill du testa detta enkla maskinkodsprogram, börja med att notera värdet på adress 49200 genom att skriva (i direktläge):

```
PRINT PEEK(49200)
```

Din dator kommer förmodligen att svara `0` eller `255`, följt av `READY.` Mata sedan in programmet genom att skriva följande:

```
A=49152
POKE A,169
POKE A+1,7
POKE A+2,141
POKE A+3,48
POKE A+4,192
POKE A+5,96
```

Mellan varje kommando kommer din dator att svara: `READY.`

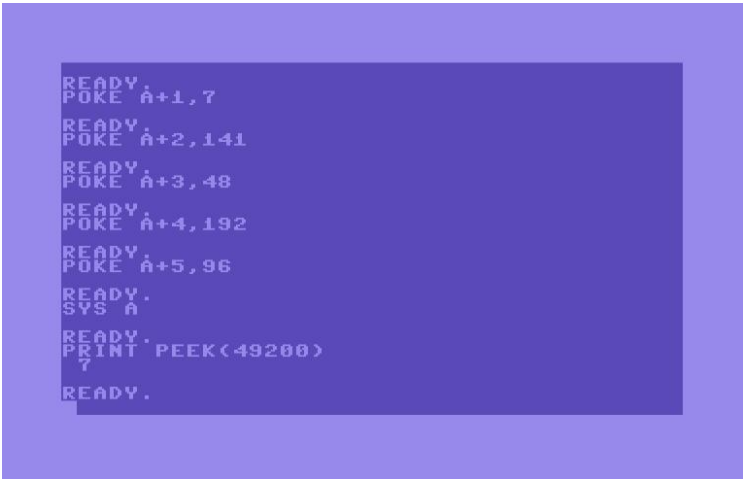
Kommandot `SYS` används för att exekvera maskinkodsprogram. Som argument skickar man startadressen. Det innebär att vi skriver följande för att starta programmet:

```
SYS A
```

Om allt fungerar som det ska, borde datorn svara med endast:

```
READY.
```

Och när du på nytt läser av minnesadress 49200 borde du få svaret `7`, enligt bilden.



Figur 43: Inmatning av maskinkodsprogram som sparar värdet 7 på adress 49200.

Låt oss skapa ett program på adress 49152 som summerar två tal, 10 och 15, och sparar resultatet (25) i minnet på adress 49200. Operationskoden `105` adderar ett värde till det som redan ligger i ackumulatorn.

```
169 10
105 15
141 48 192
96
```

I assembler skulle programmet kunna uttryckas så här:

```
LDA #10
ADC #15
STA 49200
RTS
```

Programmet startas med `SYS 49152` och resultatet läses av med `PRINT PEEK(49200)`. Du borde få svaret `15`.

Detta är den mest grundläggande principen med maskinkod. Många funktioner i din Commodore-dator, bygger på att olika värden sätts i olika minnesadresser, så förfarandet att sätta värden är väldigt centralt, och vilka värden som har vilken effekt beror på vilken dator du sitter med.

Om du använder VIC-20, kan du skriva ett program på adress `828`, som sätter värdet `7` på adress `900`. Programmet ser då ut så här:

```
169 7
141 132 3
96
```

Starta programmet med `SYS 828` och läs av resultat med `PRINT PEEK(900)`. Och programmet som summerar `10` och `15` ser ur så här:

```
169 10
105 15
141 132 3
96
```

En maskinkodsmonitor är en mjukvara som låter dig skriva in operationerna och argumenten som text. På det viset kan du skriva `LDA` i stället för (till exempel) `169`. Om du äger en Commodore 128 finns en sådan inbyggd, som du når genom att skriva `MONITOR`. Om du äger en Commodore 64 rekommenderar jag The Final Cartridge III, och om du äger en VIC-20 så rekommenderar jag VICMON. Dessa tre mjukvaror fungerar lite olika, så även om du är bekant med den ene bör du läsa manualen för den andre, när du växlar mellan dem.

Vad gör programmet som presenteras i avsnittet om DATA?

I avsnittet om `DATA` i kapitlet om kommandon så presenteras en *loader* som skriver ett maskinkodsprogram till minnet. Programmet lyder som följer:

```
169 5 141 33 208 96
```

Från det vi vet om operationskoder och argument, så kan vi strukturera programmet så här, för läsbarhet:

```
169 5
141 33 208
96
```

Och översätter vi detta till vad operationskoderna betyder, så får vi:

```
LDA 5
STA 33 208
RTS
```

Och om vi applicerar det vi vet om 16-bitarstal så får vi:

```
LDA 5
STA 53281
RTS
```

Värdet `5` lagras alltså på adress `53281`, som beskriver bakgrundsfärgen, och där betyder `5` grön.

INDEX

Index

BILDER

Bilder

Erkännanden:

Ett stort tack till de som läst, hittat fel och påpekat dessa för mig. Tack till Alvaro Alonso, som lärde mig ett och annat om prestandaskillnaderna i `FOR` och `GOTO`.