

*Commodore BASIC 7.0*

*för Commodore 128*

*av Anders Hesselbom*

Commodore BASIC 7.0

för Commodore 128  
av Anders Hesselbom

Publicerad 2022-XX-XX, revision 0, 2022-XX-XX. Rättningar och kommentarer skickas till anders@winsoft.se

Innehåll

[Introduktion 3](#_Toc113997273)

[Om Commodore 128 12](#_Toc113997274)

[Kommandon 14](#_Toc113997275)

[Funktioner 49](#_Toc113997276)

[Sprites 65](#_Toc113997277)

[Text 67](#_Toc113997278)

[Grafik 69](#_Toc113997279)

[Ljud 71](#_Toc113997280)

[Musik 73](#_Toc113997281)

[Ljudeffekter 75](#_Toc113997282)

[80-kolumnsläge 77](#_Toc113997283)

[Commodore 64-läge 79](#_Toc113997284)

[CP/M 81](#_Toc113997285)

[Appendix A: Ordförklaringar 83](#_Toc113997286)

[Appendix B: Commodore BASIC 2.0 second release 85](#_Toc113997287)

[Appendix C: En jämförelse mellan Commodore 128, Commodore 64 och VIC-20 87](#_Toc113997288)

[Appendix D: Maskinkod 89](#_Toc113997289)

[Index 91](#_Toc113997290)

[Bilder 95](#_Toc113997291)

KAPITEL 1: INTRODUKTION

# Introduktion

XXX

## 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. XXX BYT BILD



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 7.0

XXX

## 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

XXX Commodore BASIC 7.0 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 nyckelord 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 nyckelord 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$).

Utöver nyckelorden finns en uppsättning operatorer som skrivs som tecken. Dessa är jämförelseoperatorerna och beräkningsoperatorerna.

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]](#footnote-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.

## Om textkonsolen

XXX

**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

XXX

## Bokens innehåll

XXX

KAPITEL 2: OM COMMODORE 128

# Om Commodore 128

XXX

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[[2]](#footnote-2).

|  |  |
| --- | --- |
| 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.*

## 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)[[3]](#footnote-3).

|  |  |
| --- | --- |
| 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[[4]](#footnote-4).

|  |  |
| --- | --- |
| 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*.

## 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[[5]](#footnote-5).

|  |  |
| --- | --- |
| 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[[6]](#footnote-6).

|  |  |
| --- | --- |
| 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.

## 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[[7]](#footnote-7).

|  |  |
| --- | --- |
| 10 PRINT "A" 20 END 30 PRINT "B" | Rad 10 skriver A på skärmen.  Rad 20 avbryter exekveringen.  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*.

Följande program tillhandahåller tre heltalskonstanter. Två av dessa blir lästa in i variabler, och sedan utskrivna på skärmen[[8]](#footnote-8).

|  |  |
| --- | --- |
| 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)[[9]](#footnote-9):

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[[10]](#footnote-10).

## 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[[11]](#footnote-11):

|  |  |
| --- | --- |
| 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 *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[[12]](#footnote-12).

|  |  |
| --- | --- |
| 10 DIM F%(9)  20 A%=1  30 B%=0  40 FOR I=0 TO 9  50 F%(I)=A%+B%  60 A%=B%  70 B%=F%(I)  80 NEXT  90 FOR I=0 TO 9  100 PRINT F%(I)  110 NEXT | Rad 10 skapar en heltalsarray, F%, med 10 element (0 till 9).  Rad 20 initierar A% med värdet 1.  Rad 30 initierar B% med värdet 0.  Rad 40 påbörjar en iteration från 0 till 9.  Rad 50 lägger summan av A% och B% i ett element i arrayen F%.  Rad 60 kopierar värdet av B% till A%.  Rad 70 kopierar värdet av aktuellt F%-element till B%.  Rad 80 stänger iterationen.  Rad 90 påbörjar en iteration från 0 till 9.  Rad 100 skriver ut värdet ett värde från ett element i arrayen F%.  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[[13]](#footnote-13).

|  |  |
| --- | --- |
| 10 PRINT "A"  20 END  30 PRINT "B" | Rad 10 skriver ut A på skärmen.  Rad 20 avslutar programmet.  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 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*.

Se kapitlet om användardefinierade funktioner för mer information om FN.

## 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[[14]](#footnote-14).

|  |  |
| --- | --- |
| 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[[15]](#footnote-15).

|  |  |
| --- | --- |
| 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. |

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[[16]](#footnote-16):

|  |  |
| --- | --- |
| 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[[17]](#footnote-17).

|  |  |
| --- | --- |
| 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). |

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[[18]](#footnote-18).

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[[19]](#footnote-19):

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.

## GET

Kommandot GET används i runtime för att läsa från tangentbordet. I direkt-läge 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  20 PRINT A | Rad 10 ökar A med 1.  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  20 PRINT A  30 GOTO 10 | Rad 10 ökar A med 1.  Rad 20 skriver ut värdet av A.  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.

En bild som visar skärmbild

Automatiskt genererad beskrivning

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

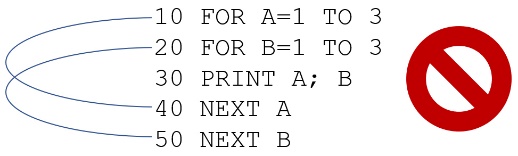
NEXT stänger en FOR-iteration. Följande exempel exekverar rad 20 fem gånger:

|  |  |
| --- | --- |
| 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[[20]](#footnote-20):

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[[21]](#footnote-21):

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.

PRINT kan skrivas som ?, vilket innebär att både PRINT 4+5 och ? 4+5 svarar på vad summan av 4 och 5 är.

## 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.

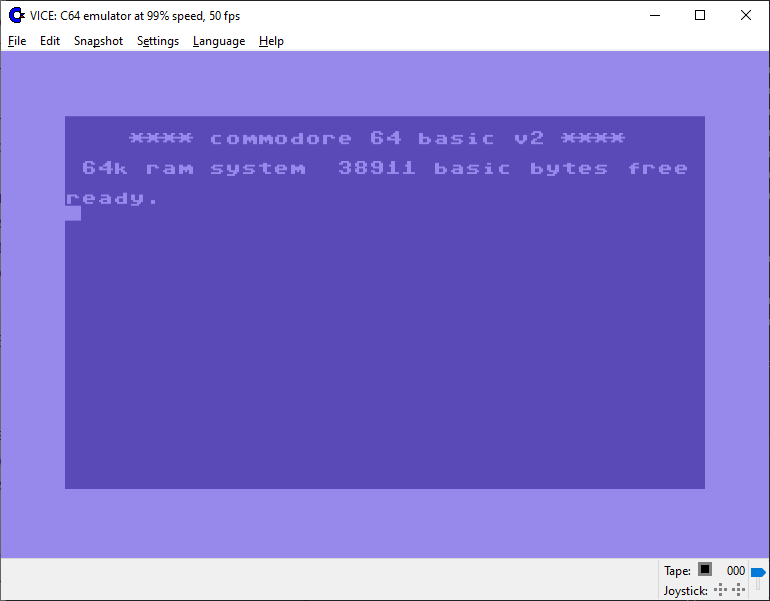
10 PRINT "HEJ":REM SKRIV HEJ

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:

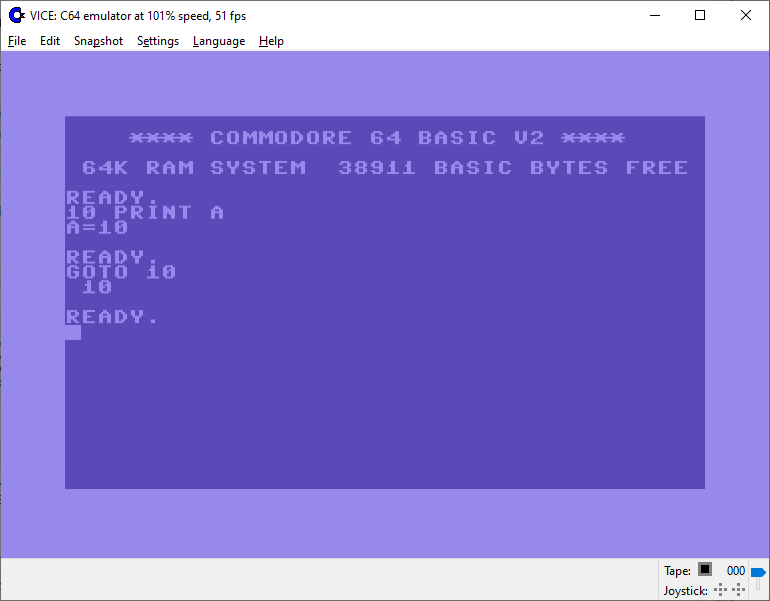
A  
B  
C  
D  
E

## 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*.

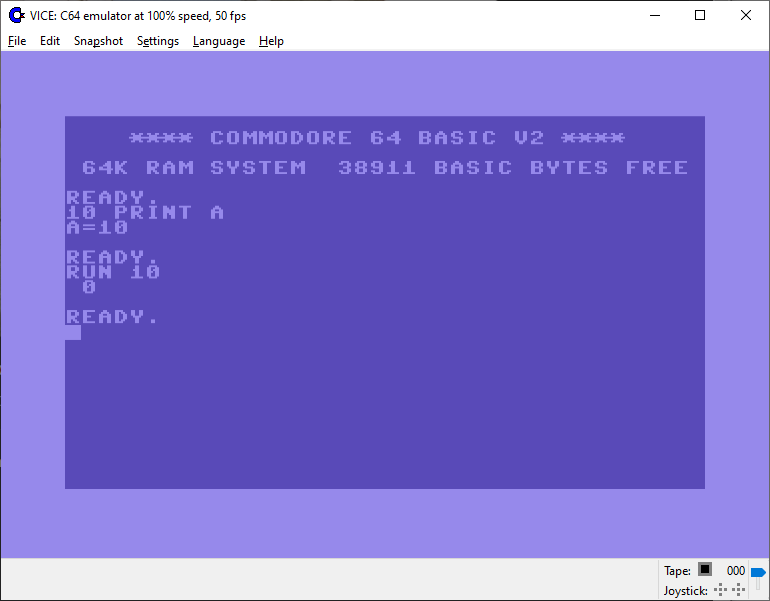
Det går att använda RUN i runtime för att göra hopp, vilket ger samma effekt som att kombinera CLR med GOTO.

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)

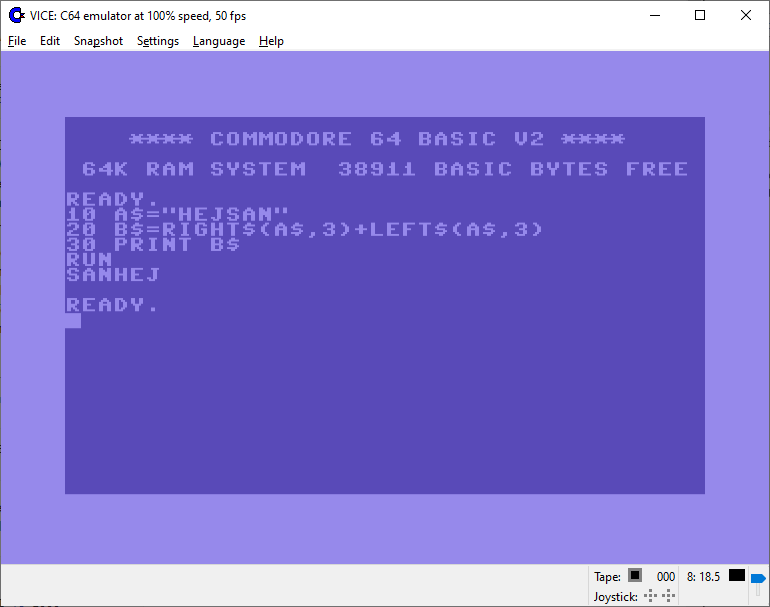
Du kan testa genom att skriva PRINT A i direkt-läge. Din dator svarar med att skriva ut 4 på skärmen.

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):

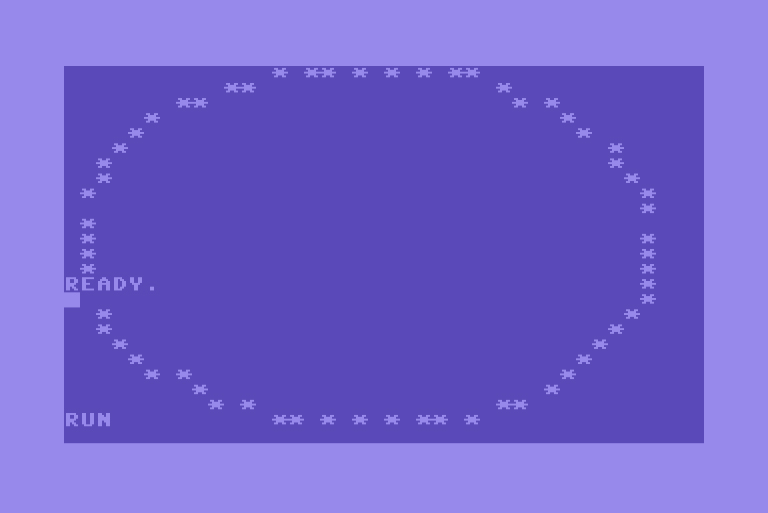
|  |  |
| --- | --- |
| 10 FOR I=0 TO 63  20 X=INT(COS(I/10)\*18)+19  30 Y=INT(SIN(I/10)\*12)+12  40 POKE 780,0  50 POKE 782,X  60 POKE 781,Y  70 SYS 65520  80 PRINT "\*"  90 NEXT | Rad 10 startar en iteration som ska köras 64 gånger.  Rad 20 och 30 hämtar nästa cosinus- och sinus-tal, och skalar upp 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. 103 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.

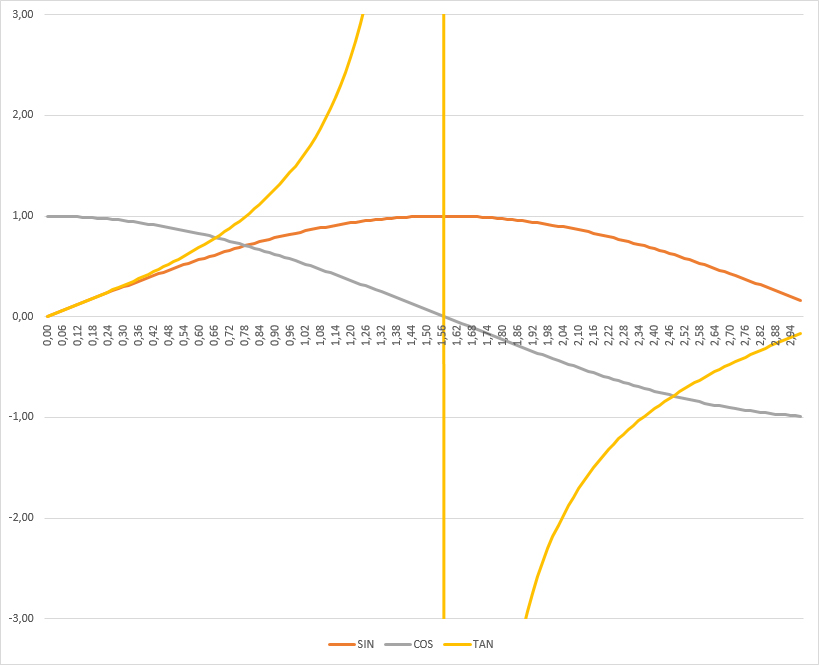
* PRINT 1 TAB(5) "2" ger 1 2 (med inledande blanksteg före 1)
* PRINT 11 TAB(5) "2" ger 11 2
* PRINT 111 TAB(5) "2" ger 111 2 (ett blanksteg mellan 111 och 2)
* PRINT 1111 TAB(5) "2" ger 1111 2 (även här ett blanksteg mellan, vilket är så nära den avsedda platsen som möjligt)

## 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
* VAL accepterar strängar som beskriver ett tal uttryckt med *vetenskaplig notation* (se nedan)
* Argument som inte är textsträngar orsakar felet *type mismatch*
* Strängar som representerar för stora eller för små tal orsakar felet *overflow*

Observera att följande exempel syftar till att visa hur man konverterar från text till tal med VAL. Exemplet visar inte hur man på ett korrekt sätt samlar in numerisk data från användaren[[22]](#footnote-22):

|  |  |
| --- | --- |
| 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

Vetenskaplig notation används för att beskriva mycket stora tal utan att konsumera mer minne än ett vanligt flyttal. Konventionen på VIC-20 och Commodore 64 är som följer: Talets mantissa (decimaldelen av ett tal), bokstaven E och talets exponent (alltså vad mantissan är upphöjd med). Om du vill se hur det ser ut, kan du be din dator att skriva ut talet en miljard (en etta följt av nio nollor).

PRINT 1000000000

Detta ger svaret 1E+09. Notera att exponenten alltid har en signatur (+ eller -) som i detta fall är +. För att räkna ut vad 1E+09 faktiskt betyder, ersätt E med \*10. 1\*109 (eller 109) är en miljard.

Så vilket värde ger följande uttryck om vi skickar det till VAL? Glöm inte att punkt fungerar som decimalavgränsare.

PRINT VAL("2.5E+9")

Rätt svar är 2,5 miljarder, eftersom 2,5\*109 är 2,5 miljarder.

KAPITEL 5: SPRITES

# Sprites

XXX

KAPITEL 6: TEXT

# Text

XXX

KAPITEL 7: GRAFIK

# Grafik

XXX

KAPITEL 8: LJUD

# Ljud

XXX

KAPITEL 9: MUSIK

# Musik

XXX

KAPITEL 10: LJUDEFFEKTER

# Ljudeffekter

XXX

KAPITEL 11: 80-KOLUMNSLÄGE

# 80-kolumnsläge

XXX

KAPITEL 12: COMMODORE 64-LÄGE

# Commodore 64-läge

XXX

KAPITEL 13: CP/M

# CP/M

XXX

APPENDIX A: ORDFÖRKLARINGAR

# Appendix A: Ordförklaringar

xxx

APPENDIX B: COMMODORE BASIC 2.0 SECOND RELEASE

# Appendix B: Commodore BASIC 2.0 second release

XXX

APPENDIX C: EN JÄMFÖRELSE MELLAN COMMODORE 128, COMMODORE 64 OCH VIC-20

# Appendix C: En jämförelse mellan Commodore 128, Commodore 64 och VIC-20

XXX

APPENDIX D: MASKINKOD

# Appendix D: Maskinkod

XXX

INDEX

# Index

?, 38

16-bitarstal, 75

ABS, 50

array, 19

ASC, 51

ATN, 51

CHR$, 52

CLOSE, 14

CLR, 15

CMD, 16, 31

CONT, 17

COS, 52, 60

DATA, 17

DEF, 19

DIM, 19

direktläge, 6

diskdrive, 25

enhetscirkeln, 52, 57

EXP, 53

exponent, 62

fibonaccisekvens, 20

FN, 21

FOR, 21

FRE, 53

fysisk fil, 35

GET, 25

GET#, 25

GOSUB, 26, 34

GOTO, 27, 34

grad, 52

IF, 27, 45

INPUT, 28

INPUT#, 30

INT, 53

kodkommentarer, 38

LEFT$, 54

LEN, 54

LET, 30

LIST, 31

LOAD, 31

loader, 18

LOG, 54

logisk fil, 35

mantissa, 62

MID$, 54

NEW, 32

NEXT, 21, 33

ON, 34

OPEN, 35

operand, 6

POKE, 36

POS, 55

PRINT, 37

PRINT#, 38

radian, 51

REM, 39

RESTORE, 40

RETURN, 26, 41

RIGHT$, 56

RND, 56

RUN, 42

runtime-läge, 6

SAVE, 43

sekventiell fil, 14

SGN, 57

SIN, 57, 60

SPC, 58

SQR, 59

STEP, 22

STOP, 45

STR$, 59

SYS, 45

TAB, 60

TAN, 60

tangent, 51

THEN, 45

VAL, 62

VERIFY, 46

vetenskaplig notation, 62

WAIT, 46

BILDER

# Bilder

[Figur 1: Tangentbordslayout på Commodore 64. 3](#_Toc113997336)

[Figur 2: Språkets utveckling. 5](file:///C:\GitRepos\CommodoreBASIC70\Commodore%20BASIC%207.0%20for%20C128.docx#_Toc113997337)

[Figur 5: INPUT säkerställer att användaren skriver in ett korrekt värde. 30](#_Toc113997338)

[Figur 6: Här visas hur NEW tömmer både variabel- och BASIC-minnet. 32](#_Toc113997339)

[Figur 7: Effekten av kommandot NEW i ett program. 32](#_Toc113997340)

[Figur 8: Felaktigt nästlande. 33](#_Toc113997341)

[Figur 9: Tryck på Commodore+Shift för att växla mellan versaler och grafik eller gemener och versaler. 39](#_Toc113997342)

[Figur 10: Läget för gemener och versaler. 40](#_Toc113997343)

[Figur 11: När ett program startas med GOTO bevaras alla variabler. 42](#_Toc113997344)

[Figur 12: När ett program startas med RUN rensas alla variabler. 42](#_Toc113997345)

[Figur 13: Ett program som bygger en sträng av de tre sista och de tre första tecknen i en befintlig sträng. 56](#_Toc113997346)

[Figur 14: En cirkel skapad med COS och SIN på Commodore 64. 58](#_Toc113997347)

[Figur 15: Svar från funktionerna SIN, COS och TAN efter input 61](#_Toc113997348)

**Erkännanden:**

Omslagsbilden föreställande en Commodore 128 är fotograferad av Evan Amos (CC BY-SA 3.0).

1. När man bygger en IF-sats kan det vara nödvändigt att använda kolon, vilket förklaras under avsnittet om IF i kapitlet om kommandon. [↑](#footnote-ref-1)
2. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/close.bas [↑](#footnote-ref-2)
3. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/clr1.bas [↑](#footnote-ref-3)
4. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/clr2.bas [↑](#footnote-ref-4)
5. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/cmd1.bas [↑](#footnote-ref-5)
6. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/cmd2.bas [↑](#footnote-ref-6)
7. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/cont.bas [↑](#footnote-ref-7)
8. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/data1.bas [↑](#footnote-ref-8)
9. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/data2.c64.bas [↑](#footnote-ref-9)
10. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/data3.vic20.bas [↑](#footnote-ref-10)
11. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/def.bas [↑](#footnote-ref-11)
12. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/fibonacci.bas [↑](#footnote-ref-12)
13. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/end.bas [↑](#footnote-ref-13)
14. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/for1.bas [↑](#footnote-ref-14)
15. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/for2.bas [↑](#footnote-ref-15)
16. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/for3.bas [↑](#footnote-ref-16)
17. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/for4.bas [↑](#footnote-ref-17)
18. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/speedtest.goto.c64.bas [↑](#footnote-ref-18)
19. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/speedtest.for.c64.bas [↑](#footnote-ref-19)
20. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/speedtest.poke.constant.bas [↑](#footnote-ref-20)
21. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/speedtest.poke.var.bas [↑](#footnote-ref-21)
22. https://github.com/Anders-H/CommodoreBASIC20/blob/main/Source/val.bas [↑](#footnote-ref-22)