TRICKS & TIPS FOR THE COMMODORE

YOU CAN COUNT ON
Abacus Software

TRICKS & TIPS
FOR THE COMMODORE 64

BY: Klaus Gerits
Lothar Englisch
Michael Angerhausen

A DATA BECKER BOOK

Abacus Software

P.O. BOX 7211 GRAND RAPIDS, MICH. 49510

Second English Edition, Nov 1984

Printed in U.S.A.

ISBN 0-916439-03-8

Table of Contents

4. Advanced BASIC

5. Forth: An alternative to BASIC

6. CP/M on the Commodore 64

Chapter 1 : Introduction

The Commodore 64 wins thousands of new friends every day all over the world. That is hardly surprising since the 64 offers not only excellent performance, but also an excellent price to performance ratio. One can now purchase a Commodore 64 complete with disk drive for under 500 dollars. The 64 carries the price of an introductory computer, but it offers far more than just game playing or an introduction to computing. It offers the hobbyist an almost boundless tool with which to work and can also be used for small business and scientific applications.

Here then is Tricks & Tips, our fourth book for the Commodore 64. Our experienced team of authors consisting of Klaus Gerits, Lothar Englisch, and Michael Angerhausen has again filled this book with programming tricks. The authors hope to provide ideas for your own programs through the use of countless examples and model programs. This book is intended to help you, the programmer, get more out of your Commodore 64.

Chapter 2 : Advanced Graphics

2.1 Graphics on the Commodore 64

Sooner or later every Commodore 64 user has the desire to work with the built-in graphics capabilities of this computer. Unfortunately, the instruction manual says little about the capabilities and possibilities that the Commodore 64 offers.

At this point, we want to take a more detailed look at the graphics possibilities and features.

First one must distinguish between the normal graphics, i.e. the symbols of which are shown on the keys (the block or line graphics), the high-resolution graphics, and the sprites. Some computers offer block graphics and high-resolution graphics, but the sprites are something truly new on the Commodore 64. These sprites were previously found only on video arcade games. And now these same capabilities are offered to us by the Commodore 64.

On the next pages we want to go over the three graphics modes. We will of course help you by illustrating the theory with many examples.

2.1.1 The block or line graphics on the keyboard

This method of creating graphics on the Commodore 64 is the simplest and easiest. No addresses have to be calculated nor attention paid to any registers. One can create graphics directly from the keyboard and place them in the program while both are being developed. It is usually necessary to press two keys to obtain these symbols. If you look at the keyboard closely, you will see that almost every key has two graphics symbols on it in addition to the normal letter. The symbol or graphics character on the left side of the key is obtained by first pressing the Commodore (C=) key, holding this down, and then pressing the corresponding key with the desired graphics character.

These characters can always be entered within a PRINT or INPUT statement. One might write

```
100 PRINT "
```

for example, and then press the keys C= and A. You now see the upper right-hand corner of a frame on the screen. To create an entire frame, the following input is necessary (still in line 100):

Press the shift and * keys 38 times. You see a straight line extending from the corner of the frame on the same horizontal line. In addition, you have also learned that you can enter the graphics character shown on the right side of a key by pressing SHIFT along with that key. Now press the keys C= and S to complete the top part of the frame. At the end of line 100, enter the following:

```
";
```

and press the RETURN key

The next line can be entered as follows:

```
110 PRINT "
```

After this, press SHIFT and - one time, the space bar 38 times, and SHIFT and - once again. At the end, enter

```
";
```

and press the RETURN key again.

The second line of the frame is already done. The third and last line is written as follows:

```
120 PRINT "
```

and then C= and Z, SHIFT and * 38 times, then C= and X. Now we have the complete frame consisting of three lines. Enter these lines:

```
99 PRINT CHR$(147);: REM ERASE THE SCREEN
132 A$="                                      "
135 REM A$=38 SPACES
140 B$="-THIS LINE FILLS OUR FRAME COMPLETELY-"
150 PRINT CHR$(19);
160 PRINT CHR$(17);CHR$(29);A$;
170 PRINT CHR$(19);
180 PRINT CHR$(17);CHR$(29);B$;
190 FOR I=1 TO 1000: NEXT
200 GOTO 150
```

Today, many commercial programs use such frames to make the screen appear more professional and less cluttered.

Naturally, there is another way of entering such graphics symbols. These symbols can all be obtained via the CHR$ function. Here is an example using our last program:

```
100 A1$=CHR$(176): A2$=CHR$(174)
101 REM THE LEFT AND RIGHT-HAND UPPER CORNERS
102 A3$=CHR$(173): A4$=CHR$(189)
103 REM THE LEFT AND RIGHT-HAND LOWER CORNERS
104 H1$=CHR$(96)
105 REM HORIZONTAL LINE
106 H2$=CHR$(125)
107 REM PERPENDICULAR LINE
108 H3$=CHR$(32)
109 REM SPACE
110 Z1$=A1$
111 FOR I=1 TO 38
112 Z1$=Z1$+H1$
113 NEXT I
114 Z1$=Z1$+A2$
```

```
115 REM FIRST LINE OF THE FRAME
116 Z2$=H2$
117 FOR I=1 TO 38
118 Z2$=Z2$+H3$
119 NEXT I
120 Z2$=Z2$+H2$
121 REM SECOND LINE OF THE FRAME
122 Z3$=A3$
123 FOR I=1 TO 38
124 Z3$=Z3$+H1$
125 NEXT I
126 Z3$=Z3$+A4$
127 REM THIRD LINE OF THE FRAME
128 PRINT Z1$;
129 PRINT Z2$;
130 PRINT Z3$;
```

When you enter these new lines and run the program, you will get the same result as before. The advantage lies in the fact that programs written with such CHR$ functions are easier to read and change.

2.1.2 The use of sprites

Your Commodore 64 can do more that just draw simple lines or frames. It offers you graphics capabilities that we have only begun to describe, capabilities previously found only on coin-operated video games.

The Commodore 64 has eight movable graphics objects called "sprites." Each of these eight sprites can be moved, erased, or redefined via POKE commands, independent of the other sprites. In order to get the most out of sprite graphics, one must be acquainted with the corresponding registers in the Commodore 64. The complete register layout can be found in the book The Anatomy of the Commodore 64.

There are a variety of registers at our disposal for each sprite. It would be advantageous if you first experiment with the sprites which you find in the Commodore User's Guide.

The most important address to keep in mind when working with sprites is located at 53248. The built-in VIC (VIdeo Controller) 6569 has a set of registers which are mapped to addresses starting here. In order to position a sprite, for instance, we must tell the VIC chip where we want it to draw the sprite. The register we use is register 0 (having an address exactly equal to 53248). In this address we find the horizontal position and in register 1 (53249) the vertical position of the sprite.

```
POKE 53248,160: POKE 53249,120
```

Two POKEs suffice to set an entire sprite at a certain location on the screen. These two POKEs place the sprite in the middle of the screen. Registers 0 and 1 serve for sprite 1, registers 2 and 3 for sprite 2, and so on. Almost all of the registers work on this principle. Exact information about the manipulation of sprites can be found in The Anatomy of the Commodore 64. In the next pages you will learn how to create complex graphic images with minimum effort.

2.2 3D Graphics - BASIC Program

At the beginning of this book we want to present you with a BASIC program that displays three-dimensional representations of functions on the screen with the help of the Commodore 64's high-resolution graphics. The program uses the commands of a BASIC extension called **ULTRABASIC-64**; at the end you will find the necessary changes if you do not have **ULTRABASIC-64** at your disposal.

This program draws the function defined in line 100. The function can be drawn in one of three different ways:

First, the function can be shown in a normal Cartesian (rectangular) coordinate system, in the same way you would draw the function on graph paper. Second, it is possible to represent the function in the polar coordinate (radius and length) system. Third and most interesting is three-dimensional representation. The function is rotated about the (vertical) Y-axis. Because of the large number of points that must be calculated, this method requires the greatest amount of time.

Now a description of the program itself. First, you can select the means of representation (lines 40-70). For the Cartesian and polar plots you are asked for the function increment (line 260). This is the value by which the parameter of the horizontal axis is incremented after each calculation. Lines 270 and 280 ask for the scaling factors for the X and Y axes. This allows you to control the aspect ratio of the axes as well as the "magnification" factor. For the time being, enter 1 for both. Use zero for the horizontal and vertical displacements (lines 370-410). The graphics mode is selected in line 430. The lines 450 to 560 draw the axes and scales. The lines 680 to 790 draw the polar representation and lines 820 to 970 draw the rectangular graph.

The three-dimensional plotting routine starts at line 1010. You can again select the values for scale and position. For now, enter the suggested values of 20 and 90. The three-dimensional representation requires that the function in line 100 be calculated more than ten thousand times; the program takes between one half of an hour to several hours to do this.

Run the program with various functions. Here are some functions which will yield interesting graphs.

```
100 DEF FN R(Q) = COS (2*Q) + COS ((Q + BB)/16)
100 DEF FN R(Q) = SQR (ABS(.5*(16-Q*Q)) + 1/(Q+4)
100 DEF FN R(Q) = COS (4*Q) + 20/(Q*Q + 3)
```

If you do not have ULTRABASIC-64, you must make the following changes and additions to the program:

```
Line 5   POKE 56,32 : CLR
Line 430 and 1400  GOSUB 2000
Line 470  FOR A1=0 TO 199 : AX=F: AY=A1: GOSUB 3000 : NEXT
Line 480  FOR A1=0 TO 319 : AX=A1: AY=E: GOSUB 3000 : NEXT
Line 500  FOR A1=E-1 TO E+1: AX=YR: AY=A1: GOSUB 3000 : NEXT
Line 520  FOR A1=E-1 TO E+1: AX=XL: AY=A1: GOSUB 3000 : NEXT
Line 540  FOR A1=F-1 TO F+1: AX=A1: AY=YD: GOSUB 3000 : NEXT
Line 560  FOR A1=F-1 TO F+1: AX=A1: AY=YU: GOSUB 3000 : NEXT
Line 770  AX=XX: AY=YY: GOSUB 3000
Line 900  AX=G: AY=YY: GOSUB 3000
Line 1600 AX=X1: AY=Y1: GOSUB 3000
```

```
Line 1620 GOSUB 4000 : RETURN
Line 2000 FOR A1=8192 TO 16191 : POKE A1,0 : NEXT
Line 2010 FOR A1=1024 TO 2023  : POKE A1,16: NEXT
Line 2020 POKE 53248+17, 27+32 : POKE 53248+24, 16+8
Line 2030 RETURN
Line 3000 OY=320*INT(AY/8)+(AYAND7)
Line 3010 OX=8*INT(AX/8)
Line 3020 MA=2^((7-AX)AND7)
Line 3030 AV=8192+OY+OX
Line 3040 POKE AV,PEEK(AV) OR MA : RETURN
Line 4000 FOR A1=Y1+1 TO 199:AX=X1:AY=A1:GOSUB 5000:RETURN
Line 5000 OY=320*INT(AY/8)+(AYAND7)
Line 5010 OX=8*INT(AX/8)
Line 5020 MA=2^((7-AX)AND7)
Line 5030 AV=8192+OY+OX
Line 5040 POKE AV,PEEK(AV) AND (255-MA): RETURN
```

Programming the graphics functions in BASIC makes the program considerably slower as compared to programming using ULTRABASIC-64 for example.

```
10 PRINT"{CLR}{C/DN}  GRAPHIC REPRESENTATION OF FUNCTIONS{C/
DN}"
20 PRINT "  DEFINED IN LINE 100{C/DN}"
40 PRINT"{C/DN}  1 - CARTESIAN PLOT"
50 PRINT"{C/DN}  2 - POLAR COORDINATES"
60 PRINT"{C/DN}  3 - 3D PLOT"
70 INPUT"{C/DN} CHOICE: 1{C/LF}{C/LF}{C/LF}";PL
100 DEF FNR(Q)=COS(Q)+COS(2*Q)+COS(5*Q)
210 IF PL=3 THEN 1010
250 PRINT:PRINT
260 INPUT"FUNCTION INCREMENT =";IK
270 INPUT"{C/DN}FACTOR FOR X-AXIS =";S1
280 INPUT"{C/DN}FACTOR FOR Y-AXIS =";S2
370 PRINT"LEFT OR RIGHT SHIFT"
380 INPUT"NUMBER FROM -130 TO 130 ";C
400 PRINT"UP OR DOWN SHIFT"
410 INPUT"NUMBER FROM -90 TO 90 ";D
430 HIRES 2,2
450 E=100+D:F=160+C
470 DRAW F,0,F,199,1
480 DRAW 0,E,319,E,1
490 FOR XR=F TO 319 STEP 19*S1
500 DRAW XR,E-1,XR,E+1,1 : NEXT
510 FOR XL=F TO 0 STEP -19*S1
520 DRAW XL,E-1,XL,E+1,1 : NEXT
530 FOR YD=E TO 199 STEP 15*S2
540 DRAW F-1,YD,F+1,YD,1 : NEXT
550 FOR YU=E TO 0 STEP -15*S2
560 DRAW F-1,YU,F+1,YU,1 : NEXT
580 IF PL=1 THEN 820
610 REM POLAR PLOT
620 RD=π/180 : FOR G=0 TO 360 STEP IK : T=G*RD
710 X=FNR(T)*COS(T):Y=FNR(T)*SIN(T)
730 XX=X*(19*S1)+F : YY=-Y*(15*S2)+E
740 IF XX<0 OR XX>319 THEN 780
750 IF YY<0 OR YY>199 THEN 780
770 DOT XX,199-YY,1
780 NEXT
790 END
820 REM CARTESIAN PLOT
830 FOR G=0 TO 319 STEP IK
840 X=(G-F)/(19*S1) : Y=FNR(X)
850 YY=E-(Y*15*S2)
860 IF YY<0 OR YY>199 THEN 960
900 DOT G,199-YY,1
960 NEXT
970 END
1010 REM 3D PLOT
1020 PRINT"{CLR}{C/DN}{C/DN}{C/DN}{C/RT}{C/RT}{C/RT}VERTICAL
 ASPECT"
1030 INPUT"{C/DN}{C/DN}{C/RT}{C/RT}{C/RT}-40 TO 40, TYPICALL
Y 20 ";N1
```

```
1040 PRINT"{C/DN}{C/DN}{C/DN}{C/RT}{C/RT}{C/RT}VERTICAL OFFS
ET"
1050 INPUT"{C/DN}{C/DN}{C/RT}{C/RT}{C/RT}-50 TO 150, TYPICAL
LY 90 ";N2
1260 REM CONSTANTS A,B,C,D,E,F,G
1280 A=144:B=2.25:C=N1:D=.0327:E=160:F=N2:G=199
1400 HIRES 2,2
1410 FOR H=-A TO A STEB B
1420 AA=INT(.5+SQR(A*A-H*H))
1430 FOR BB=-AA TO AA:CC=SQR(BB*BB+H*H)*D
1440 D1=FNR(CC):DD=D1*C:GOSUB1520:NEXT:NEXT:END
1450 GOTO 1450
1520 X=BB+H/B+E:Y=DD-H/B+F
1530 X1=INT(.85*X):Y1=INT(.9*(G-Y)):IFY<0ORY>199THENRETURN
1600 DOT X1,199-Y1,1
1620 RETURN:  MODE 2 : DRAW X1,199-Y1-1,X1,0,1 : MODE 0 : RE
TURN

READY.
```

2.3 Color line graphics

The following machine language program draws vertical or horizontal lines in color. This allows data to be represented on the screen with easily understandable graphics. Because the graphics are created with the normal screen characters, text and graphics may be mixed on the screen, allowing you to label graphs, for instance. The lines are eight points wide, just like a character.

The machine language program is designed such that the length or height and color of the line can be easily controlled. The line is drawn at the current cursor position. In order to simplify the representation of a complete graphics image, the cursor is moved one position to the right after the output of a vertical line so that the next line can be drawn immediately (in a different color if necessary). After drawing a horizontal line, the cursor automatically moves one line down.

The routine is called through an expanded SYS command:

```
SYS H, L, C  or
SYS V, L, C
```

where H and V are the starting addresses for the routines to draw horizontal and vertical lines, respectively. L is the length of the line in pixels (up to 320 for a horizontal line and 200 for a vertical line), and C is the color code (0 to 15).

The machine language program begins on the following page.

```
LINE ADDR   CODE     LABEL  OPC OPERAND          COMMENTS
---- ----   ----     -----  --- -------          --------

0001 C000            ; COLOR LINE GRAPHICS
0002 C000            ; HPLOT AND VPLOT
0003 C000            ;
0004 C000            ;
0005 C000            GETCOR EQU $B7EB
0006 C000            SCROUT EQU $E716
0007 C000            LBYT   EQU $14
0008 C000            HBYT   EQU LBYT+1
0009 C000            CURCOL EQU $D3
0010 C000            SETCOL EQU $EA24
0011 C000            SETCHR EQU $EA1E
0012 C000            ILLQUA EQU $B248
0013 C000            CHKCOM EQU $AEFD
0014 C000            CODE   EQU $22
0015 C000            TMP    EQU CODE+1
0016 C000            XREG   EQU TMP+1
0017 C000            TMP1   EQU XREG+1
0018 C000            COLOR  EQU $F3              ;POINTER TO C
OLOR RAM
0019 C000            CURRIG EQU $AB3B
0020 C000            ADR    EQU $FD
0021 C000            LINELN EQU $D5
0022 C000            CHRADR EQU $D1
0023 C000                   ORG $C000
0024 C000 20 FD AE   HPLOT  JSR CHKCOM           ;COMMA
0025 C003 20 EB B7          JSR GETCOR
0026 C006 86 24             STX XREG
0027 C008 A5 15             LDA HBYT
0028 C00A C9 02             CMP #2
0029 C00C B0 42             BCS ILL
0030 C00E 0A                ASL
0031 C00F 0A                ASL
0032 C010 0A                ASL
0033 C011 0A                ASL
0034 C012 0A                ASL
0035 C013 85 23             STA TMP
0036 C015 A5 14             LDA LBYT
0037 C017 48                PHA
0038 C018 4A                LSR
0039 C019 4A                LSR
0040 C01A 4A                LSR
0041 C01B 18                CLC
0042 C01C 65 23             ADC TMP
0043 C01E 65 D3             ADC CURCOL           ;CURSOR COLUM
N
0044 C020 48                PHA
0045 C021 A8                TAY
0046 C022 C5 D3             CMP CURCOL
0047 C024 F0 13             BEQ T1
0048 C026 C9 27             CMP #39              ;<40
0049 C028 90 02             BCC T2
0050 C02A A0 27             LDY #39
```

```
0051 C02C 20 24 EA   T2      JSR SETCOL          ;POINTER TO C
OLOR RAM
0052 C02F A9 A0              LDA #$20+$80        ;REVERSE BLAN
K
0053 C031 20 1E EA           JSR SETCHR          ;SET CHAR AND
 COLOR
0054 C034 88                 DEY
0055 C035 C4 D3              CPY CURCOL
0056 C037 10 F3              BPL T2
0057 C039 68         T1      PLA
0058 C03A A8                 TAY
0059 C03B 68                 PLA
0060 C03C C0 28              CPY #40
0061 C03E B0 0B              BCS DONE
0062 C040 29 07              AND #7
0063 C042 AA                 TAX
0064 C043 BD 53 C0           LDA TABLE,X
0065 C046 A6 24              LDX XREG
0066 C048 20 1E EA           JSR SETCHR
0067 C04B A9 11      DONE    LDA #17             ;CURSOR DOWN
0068 C04D 4C 16 E7           JMP SCROUT
0069 C050 4C 48 B2   ILL     JMP ILLQUA
0070 C053 20 65 74 75 TABLE   BYT $20,$65,$74,$75
0071 C057 61 F6 EA E7         BYT $61,$F6,$EA,$E7
0072 C05B 20 FD AE   VPLOT   JSR CHKCOM
0073 C05E 20 EB B7           JSR GETCOR
0074 C061 A5 15              LDA HBYT
0075 C063 D0 EB              BNE ILL
0076 C065 86 24              STX XREG            ;COLOR
0077 C067 A5 14              LDA LBYT
0078 C069 4A                 LSR
0079 C06A 4A                 LSR
0080 C06B 4A                 LSR
0081 C06C 85 23              STA TMP
0082 C06E A5 14              LDA LBYT
0083 C070 29 07              AND #7
0084 C072 85 25              STA TMP1
0085 C074 A5 D1              LDA CHRADR          ;LINE ADDRESS
0086 C076 18                 CLC
0087 C077 65 D3              ADC CURCOL          ;PLUS CURSOR
COLUMN
0088 C079 85 FD              STA ADR
0089 C07B A5 D2              LDA CHRADR+1
0090 C07D 69 00              ADC #0
0091 C07F 85 FE              STA ADR+1
0092 C081 A0 00              LDY #0
0093 C083 A6 23              LDX TMP
0094 C085 F0 20              BEQ T3
0095 C087 20 C7 C0   T4      JSR CLR             ;CALCULATE CO
LOR ADDR
0096 C08A A9 A0              LDA #$20+$80
0097 C08C 91 FD              STA (ADR),Y
0098 C08E A5 24              LDA XREG            ;COLOR
0099 C090 91 F3              STA (COLOR),Y       ;SET CHAR AND
 COLOR
```

```
C092 A5 FD                LDA ADR
C094 38                   SEC
C095 E9 28                SBC #40              ;NEXT LINE
C097 85 FD                STA ADR
C099 B0 08                BCS T5
C09B C6 FE                DEC ADR+1
C09D A5 FE                LDA ADR+1
C09F C9 04                CMP #4               ;TOP LINE REA
CHED
C0A1 90 12                BCC T6
C0A3 C6 23       T5       DEC TMP
C0A5 D0 E0                BNE T4
C0A7 20 C7 C0    T3       JSR CLR
C0AA A6 25                LDX TMP1
C0AC BD BF C0             LDA TAB2,X
C0AF 91 FD                STA (ADR),Y
C0B1 A5 24                LDA XREG             ;COLOR
C0B3 91 F3                STA (COLOR),Y
C0B5 A5 D3       T6       LDA CURCOL
C0B7 C5 D5                CMP LINELN           ;CURSOR IN LA
ST COL?
C0B9 F0 03                BEQ T7
C0BB 4C 3B AB             JMP CURRIG           ;CURSOR RIGHT
C0BE 60          T7       RTS
C0BF 20 64 6F 79 TAB2       BYT $20,$64,$6F,$79
C0C3 62 F8 F7 E3            BYT $62,$F8,$F7,$E3
C0C7 A5 FD       CLR      LDA ADR
C0C9 85 F3                STA COLOR
C0CB A5 FE                LDA ADR+1
C0CD 4C 2A EA             JMP SETCOL+6

ASSEMBLY COMPLETE.
```

Here is a loader program in BASIC for those who do not have an assembler or monitor at their disposal.

```
100   FOR I=49152 TO 49359
110     READ X:
        POKE I,X:
        S=S+X:
      NEXT
120   DATA 32,253,174,32,235,183,134,36,165,21,201,2
130   DATA 176,66,10,10,10,10,10,133,35,165,20,72
140   DATA 74,74,74,24,101,35,101,211,72,168,197,211
150   DATA 240,19,201,39,144,2,160,39,32,36,234,169
160   DATA 160,32,30,234,136,196,211,16,243,104,168,104
170   DATA 192,40,176,11,41,7,170,189,83,192,166,36
180   DATA 32,30,234,169,17,76,22,231,76,72,178,32
190   DATA 101,116,117,97,246,234,231,32,253,174,32,235
200   DATA 183,165,21,208,235,134,36,165,20,74,74,74
210   DATA 133,35,165,20,41,7,133,37,165,209,24,101
220   DATA 211,133,253,165,210,105,0,133,254,160,0,166
230   DATA 35,240,32,32,199,192,169,160,145,253,165,36
240   DATA 145,243,165,253,56,233,40,133,253,176,8,198
250   DATA 254,165,254,201,4,144,18,198,35,208,224,32
260   DATA 199,192,166,37,189,191,192,145,253,165,36,145
270   DATA 243,165,211,197,213,240,3,76,59,171,96,32
280   DATA 100,111,121,98,248,247,227,165,253,133,243,16
      5
290   DATA 254,76,42,234
300   IF S<>26696
      THEN PRINT "ERROR IN DATA !!":
        END
310   PRINT "OK"
```

Let us take a look at a possible use for these graphics routines. This example represents sales statistics graphically.

```
100 REM THE MONTH-END TOTALS FOR THE YEAR
110 REM ARE IN THE DATA STATEMENTS
120 DIM U(12)
130 REM READ THE DATA
140 FOR I= 1 TO 12 : READ U(I) : NEXT
150 REM DETERMINE MAXIMUM VALUE
160 MAX = 0
170 FOR I= 1 TO 12
180 IF U(I) > MAX THEN MAX = U(I)
190 NEXT
200 V = 12*4096+5*16+11 : REM ADDRESS OF THE ML ROUTINE
220 PRINT CHR$(147) : REM ERASE SCREEN
230 FOR I= 1 TO 21 : PRINT CHR$(17); : NEXT : REM CURSOR
240 REM DRAW GRAPHICS
250 FOR I= 1 TO 12
260 PRINT SPC(2); : SYS V, U(I)/MAX * 180 , I
270 NEXT
280 PRINT : PRINT
290 REM WRITE MONTH NUMBER
300 FOR I=1 TO 12
310 PRINT RIGHT$(" "+STR$(I),3);
320 NEXT
330 GET A$ : IF A$="" THEN 330
400 REM SALES DATA
410 DATA 12000, 13500, 11000, 8000, 14000, 9000
420 DATA 13800, 14000, 12750, 14000, 13800, 17200
```

Now let's examine a function with horizontal line graphics.

The color code can be entered in line 100. The screen is then erased, the background color set to black, and the cursor placed in the second column. The function is calculated from -2.2 to 2.2 in lines 120 and 130, expanded to an easily representable size, and finally plotted with the SYS command.

```
100 INPUT "COLOR";C:IF C<1 OR C>15 THEN 100
110 H=12*4096 : PRINT CHR$(147)TAB(2); : POKE 53281,0
120 FOR I=-2.2 TO 2.2 STEP .2
130 SYS H,EXP(-I*I)*300,C : NEXT
```

2.4 Defining a character set

A special feature of the Commodore 64 is the ability to place the character generator in RAM. This gives you the opportunity to define your own characters.

How is a character defined?

The shape of each character is determined by something called the character matrix, an array of eight by eight pixels. Each matrix point is determined by a bit in the character generator. Each character requires 64 bits or eight bytes for a complete definition. If a bit is zero, then the corresponding point in the matrix is not set, while a set bit indicates a set point in the matrix. If a bit in the matrix is set, then it appears on the screen. The following program displays the matrix of a character on the screen. The program uses the modified PEEK function from Section 9.5--load or enter the program found there before you enter this one.

```
100 PRINT CHR$(147):PRINT:PRINT:PRINT
110 INPUT "PLEASE ENTER A CHARACTER ";A$
120 PRINT CHR$(19)A$ :B=PEEK(1024)
130 PRINT:PRINT:PRINT:PRINT
140 CG = 13*4096 : REM START OF THE CHARACTER GENERATOR
150 REM DETERMINE IF UPPER/GRAPHICS OR UPPER/LOWER MODE
160 B = (PEEK(53248+24) AND 2) * 1024
170 FOR I=0 TO 7
180 Z = USR (CG+B+8*C+I) : REM GET MATRIX A LINE AT A TIME
190 FOR J=7 TO 0 STEP -1
200 A = Z AND 2^J
```

```
210 IF A THEN PRINT "*"; : GOTO 230
220 PRINT ".";
230 NEXT
240 PRINT
250 NEXT
260 RUN
```

The program asks for the character whose matrix it should display. The ASCII code of the character is put into the variable B in line 120. Line 140 checks for upper/lower case or upper/graphics mode. In line 160 the starting position of the character definition matrix within the character generator is determined. Line 180 determines if the matrix point is set or not. An asterisk is printed if the point (bit) is set, while a period is printed if it is not. Enter a "T", for example, and you will receive this output:

```
*******.
...**...
...**...
...**...
...**...
...**...
...**...
........
```

After you have seen what the matrix of an individual character looks like, we can proceed to define or redefine our own characters. To do this, we must copy the character generator from ROM to RAM and then inform the operating system where the new character generator is. The screen memory at address $C400 (decimal 50176 to 51175) is shifted at the same time. This can be accomplished in BASIC with a POKE loop. We will again use the USR function from Section 9.5.

```
100 FOR I=13*4096 TO 14*4096-1
110 POKE I+4096,USR(I) : NEXT
120 POKE 53272,24 : POKE 56576,148 : POKE 648,196
```

After RUNning this program you can define your own characters with the following program. The program prompts you for the character to be modified. You can then enter the character matrix, thereby redefining the character that is to be displayed. An asterisk indicates a set point and a period means the point is not set. When you are finished defining characters, enter the word "END" as the character.

```
100  REM  CHARACTER DEFINITION
110  CG=14*4096:
     REM  BASE OF THE CHARACTER GENERATOR
200  INPUT "[CS][CD][CD][CD][CR][CR][CR]CHARACTER ";A$:
     IF A$="END"
     THEN END
210  PRINT "[CH]";A$
220  C=PEEK(12*4096+1024)
230  PRINT "[CD][CD][CD][CD][CD][CD][CD][CR][CR][CR][CR][CR]
     01234567"
300  FOR I=0 TO 7
310    PRINT I;:
       INPUT A$(I):
       IF LEN(A$(I))<>8
       THEN PRINT "[CU][CU]":
         GOTO 310
320  NEXT
400  B=(PEEK(53248+24) AND 2)*1024:
     REM  UPPER CASE/GRAPHICS MODE
405  AD=CG+B+C*8
```

```
410   FOR I=0 TO 7:
        Z=0
420     FOR J=0 TO 7
430       Z=Z-(MID$(A$(I),8-J,1)="*")*2^J
440     NEXT
450     POKE AD+I,Z:
        REM  CHARACTER
460     POKE AD+1024+I,255-Z:
        REM  RVS-CHARACTER
470   NEXT
480   GOTO 200
```

2.5 Modifying the character set with a joystick

For certain applications it is often desirable to have special characters available which appear immediately upon pressing a key. Such things as Greek letters, often used in mathematical formulas, fall into this category.

When you have a suitable application, you can first draw your characters in raster representation on a piece of graph paper and then POKE the appropriate values into the duplicate of the character generator, but this is rather tedious.

Here is a small program which eases the development and definition of characters. It is necessary to use a joystick in control port 1 in order to use the program.

The program makes two copies of the built-in character generator into RAM. A character is taken from the first copy and displayed as a sprite, once in regular size and again in double size so that it is easier to read.

A flashing point (which we call the microcursor and which you can move with the joystick) appears on the screen.

The desired action (drawing lines, erasing lines, or positioning the microcursor) is accomplished by pressing the fire button on the joystick. The current mode is displayed on the screen.

Once the character is designed to your satisfaction, press the F1 key. This new character is placed into the alternate character set (second copy). To accept the character and leave it unchanged, press the G key. Now you can work on the next character. After you have edited all 512 characters the program ends.

Why 512 characters?

There are 128 printable characters in each display mode, upper/lower case or upper/graphics mode. The same characters displayed in reverse bring the total up to 256. This gives a grand total of 512 characters for the two display modes. The positions of the characters within the character generator can be found on page 132 of the user's guide for the C64.

Before we discuss the program itself, you should know the significance of the variables and memory addresses used.

First the variables:

- C base address of the first duplicate of the character to be displayed next
- CD base address of the transferred character in the second copy
- CP character position counter in the range 0-511
- JB condition of the button on the joystick
- JR position of the joystick
- JS address of control port 1
- MA counter for the operating mode
- PO microcursor position within the addressed byte
- PP address of the microcursor within the sprite data of the microcursor
- PV immediate value of the byte of the sprite data, addressed by the joystick
- SB base address of the sprite data
- V base address of the video controller
- X x position of the sprites on the screen
- XJ x position of the microcursor
- Y y position of the sprite on the screen

YJ y position of the microcursor

The addresses:

56 high byte of the pointer to top of memory
648 pointer to the start of video ram
832 start address of the cassette buffer
Because the cassette buffer is not used within the program, the machine language program is placed in it.
50196 pointer for sprite 1
50170 pointer for sprite 2
53272 pointer for video ram and character generator within the video controller
56576 This location contains the two bits which determine the 16K range for the memory addressed by the video controller.

Here is a step-by-step description of the program:

10 The top of memory is lowered because the first duplicate of the character set will be loaded here.

30-233 Sprites 1 & 2 are turned on and their color is set. The sprite pointers are loaded and sprite 2 is switched to double size.
The sprite data are first erased and the sprites are positioned in the approximate middle of the screen.

1000-1010 The machine language program.
For those who are interested, here is the program in assembly language:

```
SEI            disable interrupts
LDA #$33       make character generator available
STA 1
LDA #0
STA $5F        old block-start low
STA $5A        old block-end low
STA $58        new block-end low
LDA #$D0
STA $60        old block-start high
LDA #$F0
STA $59        new block-end high
LDA #$E0
STA $5B        old block-end high
JSR $A3BF      block shift routine
LDA #$37
STA 1
CLI            allow interrupts
RET            back to BASIC
```

1020-1040 The machine language program is put into the cassette buffer and executed--two duplicates are made of the character generator.

1060 The operating system is informed of the changes made. The positions of the character generator and video RAM are changed (necessary because of the hardware). The characters you now see on the screen are already coming from copy 2.

2000-2360 The characters are converted to sprites one after the other and can be changed.

2380 After working on all of the characters, the top of memory is raised again because copy 1 is no longer needed.

4000-4070	The joystick is polled and program execution branches depending on the position of the joystick and condition of the fire button.
5000-12040	The cursor position is modified based on the position of the joystick.
13000-13200	The position of the joystick is use to modify the position of the microcursor and this point is alternately flashed on and off.
20000-20080	Characters successfully completing editing are here transferred to the second copy.

One feature of this program to note is that the modified character generator does not take up any BASIC storage space. It is placed in RAM under the kernal (that is to be taken verbatim since RAM and ROM overlap each other in the C64).

The video RAM has been moved to address 49152, an address you should keep in mind if you do any POKEing to the video RAM. It is unfortunately determined by the hardware of the Commodore 64 that the shifting of the video RAM must accompany relocation of the character generator. This problem is explained in detail in our book The Anatomy of the Commodore 64.

```
10      POKE 56,144:
        CLR
20      V=53248:
        POKE 53281,0
30      POKE V+21,6:
        POKE V+40,1:
        POKE V+41,1
40      POKE 50169,16:
        POKE 50170,16
42      POKE V+23,4:
        POKE V+29,4
50      FOR I=0 TO 62:
          POKE 50176+I,0:
        NEXT
```

The program listing:

```
55     X=150:
       Y=100
223    POKE V+4,X
226    POKE V+2,X-40
233    POKE V+16,0
320    POKE V+5,Y:
       POKE V+3,Y+19
1000   DATA 120,169,51,133,1,169,0,133,95,133,90,133,88,1
       69,208,133,96,169,240
1010   DATA 133,89,169,224,133,91,32,191,163,169,55,133,1
       ,88,96
1020   FOR I=832 TO 832+33
1030     READ A:
         POKE I,A:
       NEXT
1040   SYS 832:
       POKE 850,160:
       SYS 832
1060   POKE 53272,8:
       POKE 56576,PEEK(56576) AND 252:
       POKE 648,192
1070   PRINT CHR$(147)
2000   C=9*4096
2020   FOR CP=0 TO 511:
         PRINT CHR$(19)CP:
         SB=50176
2040     FOR I=0 TO 7
2060       POKE SB+3*I,PEEK(C+I)
2080     NEXT I
2360     C=C+8:
         GOSUB 4000:
       NEXT CP
2380   POKE V+21,0:
       POKE 56,160:
       CLR :
       END
4000   XJ=0:
       YJ=0:
       JS=56321:
       SB=50176
4020   JR=(255-PEEK(JS)) AND 15:
       JB=(255-PEEK(JS)) AND 16
4030   IF JB
       THEN MA=MA+1:
         IF MA>2
```

```
        THEN MA=0
4040  ON JR
      GOTO 5000,6000,4020,7000,8000,9000,4020,10000,1100
        0,12000
4045  IF PEEK(203)<>4
      THEN 4066
4050  PRINT CHR$(19)CHR$(145)CHR$(18)"SAVE":
      GOSUB 20000
4055  RETURN

4066  IF MA=1
      THEN PRINT CHR$(145)CHR$(145)CHR$(18)" SET"
4067  IF MA=2
      THEN PRINT CHR$(145)CHR$(145)CHR$(18)" CLR"
4068  IF PEEK(203)=26
      THEN RETURN

4069  IF MA=0
      THEN PRINT CHR$(145)CHR$(145)"    "
4070  GOSUB 13000:
      GOTO 4020
5000  REM  UP
5020  YJ=YJ-1:
      IF YJ<0
      THEN YJ=0
5040  GOSUB 13000:
      GOTO 4020
6000  REM  DOWN
6020  YJ=YJ+1:
      IF YJ>7
      THEN YJ=7
6040  GOSUB 13000:
      GOTO 4020
7000  REM  LEFT
7020  XJ=XJ-1:
      IF XJ<0
      THEN XJ=0
7040  GOSUB 13000:
      GOTO 4020
8000  REM  LEFT UP
8020  XJ=XJ-1:
      IF XJ<0
      THEN XJ=0
8040  GOTO 5000
9000  REM  LEFT DOWN
9020  XJ=XJ-1:
      IF XJ<0
      THEN XJ=0
9040  GOTO 6000
10000 REM  RIGHT
10020 XJ=XJ+1:
      IF XJ>7
      THEN XJ=7
```

```
10040 GOSUB 13000:
      GOTO 4020
11000 REM  RIGHT UP
11020 XJ=XJ+1:
      IF XJ>7
      THEN XJ=7
11040 GOTO 5000
12000 REM  RIGHT DOWN
12020 XJ=XJ+1:
      IF XJ>7
      THEN XJ=7
12040 GOTO 6000
13000 REM
13020 PP=SB+YJ*3+INT(XJ/8):
      PV=PEEK(PP)
13040 PO=XJ-INT(XJ/8)*8
13060 IF PV AND 2^(7-PO)
      THEN POKE PP,(PV AND (255-2^(7-PO))):
        GOTO 13100
13080 POKE PP,(PV OR (2^(7-PO)))
13100 IF MA=1
      THEN PV=(PV OR (2^(7-PO)))
13120 IF MA=2
      THEN PV=(PV AND (255-2^(7-PO)))
13200 FOR I=0 TO 50:
      NEXT :
      POKE PP,PV:
      RETURN

20000 REM  TRANSFER NEW CHARACTER
20010 CD=C+20472
20020 FOR I=0 TO 7
20040   POKE CD+I,PEEK(SB+3*I)
20060 NEXT I
20080 RETURN
```

2.6 Dividing the screen

There is one special feature of the video controller in the Commodore 64 which makes some very interesting effects possible, but which is also seldom heard about: the raster interrupt.

In order to clarify this feature to you, we must dig a bit deeper into how the video controller creates an image on the screen.

The screen picture is constructed from individual lines, which you can see clearly if you take a close look at it. You can also recognize that a character is made up of eight lines. The video controller has a register which always contains the screen line currently being displayed. This is register 18, and is located at address 53248+18 = 53266. If you examine the contents of this register using

```
PRINT PEEK(53266)
```

the value of the raster line displayed at the exact time the PEEK command was executed is shown. Since 25 screen images are displayed in one second, you cannot obtain these values quickly enough in BASIC and must therefore program in machine language.

Another feature of the video controller is its ability to interrupt a program just before it displays a given raster line. To program a raster interrupt you must first allow the interrupt condition to actually interrupt the microprocessor (by setting the appropriate value in the interrupt register) and then setting the raster line number

at which the interrupt is to take place (by setting the value in register 18).

Interrupt service programming must be done in machine language. The following program is a short machine language program to illustrate the use of raster interrupts.

```
ADDR    CODE     LABEL  OPC OPERAND          COMMENTS
----    ----     -----  --- -------          --------

033C             IRQOLD EQU $EA31
033C             IRQVEC EQU $314
033C             RASTER EQU $D012            ;RASTER LINE
033C             IRQREG EQU $D019            ;FLAG FOR VID
EO INTERRUPT
033C             MASK   EQU $D01A            ;VIDEO CONTRO
LLER INTERRUPT MASK
033C             BORDER EQU $D020            ;BORDER COLOR
033C             COLOR  EQU $D021            ;BACKGROUND C
OLOR
033C             ICR    EQU $DC0D            ;FLAG FOR TIM
ER INTERRUPT
033C             RETIRQ EQU $FEBC            ;RETURN FROM
INTERRUPT
033C             LINE1  EQU $FB
033C             LINE2  EQU $FC
033C             COLOR1 EQU $FD
033C             COLOR2 EQU $FE
033C                    ORG 828              ;CASSETTE BUF
FER
033C 78          SETUP  SEI
033D A9 5B              LDA #<IRQNEW
033F 8D 14 03           STA IRQVEC
0342 A9 03              LDA #>IRQNEW
0344 8D 15 03           STA IRQVEC+1
0347 A5 FB              LDA LINE1
0349 8D 12 D0           STA RASTER           ;RASTER LINE
FOR INTERRUPT
034C AD 11 D0           LDA RASTER-1
034F 29 7F              AND #$7F             ;CLEAR HIGH B
IT
0351 8D 11 D0           STA RASTER-1
0354 A9 81              LDA #$81             ;PERMIT IRQ B
Y RASTER
0356 8D 1A D0           STA MASK
0359 58                 CLI
035A 60                 RTS
```

```
0029 035B                                           ;
0030 035B AD 19 D0  IRQNEW LDA IRQREG
0031 035E 8D 19 D0         STA IRQREG               ;CLEAR INTERR
UPT FLAG
0032 0361 29 01            AND #1
0033 0363 D0 07            BNE SCREEN               ;INTERRUPT BY
 RASTER LINE?
0034 0365 AD 0D DC         LDA ICR                  ;CLEAR INTERR
UPT FLAG
0035 0368 58               CLI                      ;ALLOW RASTER
 INTERRUPT
0036 0369 4C 31 EA         JMP IRQOLD
0037 036C                                           ;
0038 036C AD 12 D0  SCREEN LDA RASTER               ;READ RASTER
LINE
0039 036F C5 FC            CMP LINE2
0040 0371 B0 0D            BCS SECOND               ;GREATER THAN
 OR EQUAL SECOND VALUE?
0041 0373 A5 FD            LDA COLOR1
0042 0375 8D 20 D0         STA BORDER               ;SET COLOR 1
0043 0378 8D 21 D0         STA COLOR
0044 037B A5 FC            LDA LINE2                ;NEXT INTERRU
PT AT 2ND LINE
0045 037D 4C 8A 03         JMP EXIT
0046 0380 A5 FE     SECOND LDA COLOR2
0047 0382 8D 20 D0         STA BORDER               ;SET COLOR 2
0048 0385 8D 21 D0         STA COLOR
0049 0388 A5 FB            LDA LINE1                ;NEXT INTERRU
PT
0050 038A 8D 12 D0  EXIT   STA RASTER               ;AT LINE 1
0051 038D 4C BC FE         JMP RETIRQ

ASSEMBLY COMPLETE.
```

```
100 FOR I = 828 TO 911
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 120,169, 91,141, 20,  3,169,  3,141, 21,  3,165
130 DATA 251,141, 18,208,173, 17,208, 41,127,141, 17,208
140 DATA 169,129,141, 26,208, 88, 96,173, 25,208,141, 25
150 DATA 208, 41,  1,208,  7,173, 13,220, 88, 76, 49,234
160 DATA 173, 18,208,197,252,176, 13,165,253,141, 32,208
170 DATA 141, 33,208,165,252, 76,138,  3,165,254,141, 32
180 DATA 208,141, 33,208,165,251,141, 18,208, 76,188,254
190 IF S <> 10678 THEN PRINT "ERROR IN DATA!!" : END
200 PRINT "OK"
```

As an example, we have developed a program which allows you to display one portion of the screen with a different background color. This allows you to emphasize one or more lines on the screen. In order to keep the program as general as possible, it allows you to select the color of the emphasized area as well as the background color of the rest of the screen with POKE commands. The raster line at which the switch to the second background color occurs can be set in the same manner. This also applies to the number of the raster line at which the switch back to the first background color is made.

This program also allows you to move a colored line with a width of a standard screen line (8 raster lines) on the screen by pressing the cursor-up and cursor-down keys. The function keys can be use to change the color of the line and the remaining background.

```
100  L1=251:
     L2=L1+1:
     C1=L2+1:
     C2=C1+1
110  L=50:
     SYS 828:
     REM  INITIALIZE INTERRUPTS
120  POKE L1,L:
     POKE L2,L+8:
     POKE C1,6:
     POKE C2,8
150  GET A$:
     IF A$=""
     THEN 150
160  IF A$=CHR$(17)
     THEN GOSUB 200
170  IF A$=CHR$(145)
     THEN GOSUB 300
180  IF A$=CHR$(133)
     THEN GOSUB 400
190  IF A$=CHR$(134)
     THEN GOSUB 500
195  GOTO 150
```

```
200   IF L<240
      THEN FOR I=0 TO 7:
           L=L+1:
           POKE L1,L:
           POKE L2,L+8:
         NEXT
210   RETURN

300   IF L>50
      THEN FOR I=0 TO 7:
           L=L-1:
           POKE L1,L:
           POKE L2,L+8:
         NEXT
310   RETURN

400   POKE C1,PEEK(C1)+1 AND 15:
      RETURN

500   POKE C2,PEEK(C2)+1 AND 15:
      RETURN
```

You can change the program to suit your own purposes by changing the raster line POKEd into memory locations 251 through 254. This allows you to change the point at which the switch to the second color occurs and the raster line at which the switch is made back to the original color (locations 251 and 252, respectively). The next two addresses, 253 and 254, contain the color codes of the first and second characters.

Raster line 50 corresponds to the upper screen border (the point where the border begins), while the beginning of the lower border corresponds to raster line 250. A screen line is divided into 8 raster lines. You can also place the border between the two different colors in the middle of a screen line.

Switching background colors is not the only effect which the raster interrupt allows. Any of the video controller parameters can be changed under interrupt control. You can, for instance, mix two graphics screens or a graphics screen and a text screen on the display using the same technique we used for the background colors. You might even try displaying different character sets at different parts of the screen all at the same time!

With this technique, you can also obtain effects which are not otherwise possible with the Commodore 64. For example, the raster interrupt makes it possible to display more than 8 sprites at one time. You can display eight sprites in the upper half of the screen. When a certain raster line is reached, you simply reset the sprite pointers and coordinates and you can display eight more sprites in the lower half of the screen. Naturally, you can also divide the screen into more than two parts.

2.7 Smooth scrolling

Scrolling is the term given to the action the screen performs when all of the information on it is moved in one direction (generally up). When the screen scrolls up, a line is left blank at the bottom so that more information can be printed.

By "smooth" scrolling we mean the ability to display a new line on the screen gradually while the old line gradually disappears. The video controller allows us this possibility using register 17. The three least-significant bits allow the screen to scroll up to eight raster lines at a time, which corresponds exactly to a screen line. In order to display a new line on the screen, we can tell the video controller to display only 24 lines. This is the case when bit 3 of register 17 is cleared to zero.

First we switch the screen over to 24 lines and then position the rest of the screen contents so that the upper 24 lines will be displayed. Now we can write something to the invisible 25th line and shift the visible portion of the screen up 8 raster lines = 1 screen line. This causes the top line to disappear.

In addition to scrolling up (or down), the video controller is able to smooth-scroll horizontally, to the right or left. The three least-significant bits of register 22 apply to the column-wise shifting, while bit 3 forces the controller to display in 38 columns.

This example program scrolls the screen up.

```
20      FOR J=1 TO 100:
        NEXT
100     VIDEO=53248
110     LINE=VIDEO+17
115     X$=CHR$(19):
        FOR I=1 TO 24:
          X$=X$+CHR$(17):
        NEXT
120     POKE LINE,PEEK(LINE) AND NOT 8
130     POKE LINE,PEEK(LINE) AND 248 OR 7
140     N=N+1:
        A$="LINE"+STR$(N):
        GOSUB 200:
        GOTO 140
200     PRINT :
        PRINT X$A$;
210     FOR I=7 TO 0 STEP -1
220       POKE LINE,PEEK(LINE) AND 248 OR I
230       FOR J=1 TO 250:
          NEXT
240     NEXT :
        RETURN
```

115 A string is defined consisting of one "cursor-home" and 24 "cursor-down" characters for positioning the cursor in the 25th line.

120 Bit 3 in register 17 of the video controller is erased, switching the display to 24 lines.

130 Bits 0 through 2 are set. This displays the upper 24 lines of the screen while the 25th remains invisible at the lower screen border.

140 The counter N is incremented. The text for the line to be printed is placed in A$ for the

	subroutine at 200, and this subroutine is called.
200	The screen is shifted up one line by the PRINT command. The text is then printed on the last line.
210-240	This loop shifts the screen up 8 raster lines. The delay loop controls rate at which the scrolling will occur.

2.8 Changing the keyboard layout

The keyboard of the Commodore 64 is organized as a matrix with eight rows and eight columns. The lines of the eight rows are tied to port A (address $DC00 = 56320) of CIA 1 and the eight columns are connected to port B (address $DC01 = 56321) of CIA 1. When polling (reading) the keyboard, (address $FF9F = 65485) it is polled row by row, during which each row sends a signal over port A. If a key is pressed, you can determine the column of the pressed key over port B. The key numbers between 0 and 63 are calculated from the row and column numbers. 64 indicates that no key is pressed. The organization is given in the table below. This key number is placed in location $CB (203) after each polling. The number of the key last pressed is stored in $C5 (197). The status of the special keys is stored in address $028D = 653 when polling. Bit 0 indicates SHIFT, bit 1 is reserved for the COMMODORE key, and bit 2 is for the CTRL key. The assignment of a particular character to a particular key is controlled by various tables which determine the ASCII value to be assigned to any given key. Because all keys on the Commodore 64 can have four different meanings, there are four such tables. Notice the difference between the right and left shift keys. Shift lock is tied to the left shift key.

Col / Row	0	1	2	3	4	5	6	7
0	DEL	RETURN	CURRIGHT	F7	F1	F3	F5	CURDOWN
1	3	W	A	4	Z	S	E	SHIFT LT
2	5	R	D	6	C	F	T	X
3	7	Y	G	8	B	H	U	V
4	9	I	J	0	M	K	O	N
5	+	P	L	-	.	:	@	,
6	POUND	*	;	HOME	SHIFT RIGHT	=	^	/
7	1	ARROW	CTRL	2	SPACE	C=	Q	STOP

The first assignment table gives the ASCII code when the key is pressed alone. The second table contains the codes for when the key is pressed along with the SHIFT key, the third table for when the Commodore key is pressed, and the fourth and final table is for the control key. An entry of $FF=255 in this table marks an illegal entry. The keys SHIFT, COMMODORE, and CTRL are handled differently; the corresponding entries in the first table are 1, 2, and 4. This status is saved in $28D=653. Bits 0, 1, and 2 correspond to these three keys.

If we want to assign a different code to a key, we must change the corresponding entry in the table. Because the table is stored in ROM, it is not possible to change it directly. The Commodore 64 has RAM as well as ROM available to it in the same address range, however, allowing the kernal to be copied to the "underlying" RAM and there changed. This can be done with a small BASIC program. At the same time, BASIC itself must also be copied into the underlying RAM.

```
100 FOR I = 40960 TO 49151 : REM COPY BASIC RAM
110 POKE I, PEEK(I) : NEXT
120 FOR I = 14*4096 TO 65535 : REM COPY KERNAL
130 POKE I, PEEK(I) : NEXT
140 POKE 1,53
```

Lines 100 to 130 copy the kernal and BASIC from ROM into the underlying RAM. The switch from ROM to RAM is made in line 140, so that the kernal is now running in RAM. Now we can proceed to change the codes of individual keys.

We need to know the addresses of the four tables:

Table 1	unshifted	$EB81	= 60289
Table 2	with shift	$EBC2	= 60354
Table 3	with Commodore	$EC03	= 60419
Table 4	with CTRL	$EC78	= 60536

If we want to change a code, we must determine the number of the key we wish to change from the matrix table. The numbering runs from 0 in the upper left-hand corner to 63 in the lower right. To find the key number, multiply the row number by 8 and add the column number. Y, for instance has the number 25 and Z the number 12. The number of the key is used as the offset to the start of the desired table.

With the help of these four tables you can define 4*64 or 256 different characters. The RESTORE key cannot be redefined since it is tied directly to the non-maskable interrupt (NMI) line of the processor. The key definition remains until STOP/RESTORE is pressed. Since these two keys switch the ROM back on. This can be prevented by changing the value for the memory configuration in RAM. This can be done in the previous program in line 150:

```
150 POKE 64982, 53
```

Instead of determining the number of the key from the matrix, one can obtain it from the program itself. This program reassigns keys and determines the appropriate key number itself:

```
100 DIM T(4): FOR I=1 TO 4: READ T(I): NEXT
110 DATA 60289,60354,60419,60536
120 FOR I=14*4096 TO 65535: POKE I, PEEK(I): NEXT
130 FOR I=40960 TO 49151 : POKE I, PEEK(I) : NEXT
140 POKE 1,53: POKE 64982, 53
1000 PRINT "PLEASE PRESS THE KEY WHICH YOU WISH TO CHANGE"
1010 GET A$: IF A$="" THEN 1010
1020 PRINT A$
1030 A = ASC(A$)
1040 FOR J=1 TO 4: T=T(T)
1050 FOR I=0 TO 63: IF PEEK(T+1) <> A THEN NEXT: NEXT
1060 PRINT "PRESS THE KEY WHICH YOU WISH TO ASSIGN"
1065 PRINT "TO THE FIRST"
1070 GET A$: IF A$="" THEN 1070
1080 PRINT A$
1090 POKE T+I, ASC(A$): GOTO 1000
```

Chapter 3: Easy Data Entry

3.1 Cursor positioning and determining cursor position

For easy input and output on the screen, it is very useful to be able to set the cursor directly to any desired spot on the display. The Commodore 64 has a command for positioning the cursor on a line, the TAB command and the POS function for determining the column, but no commands for moving the cursor directly to any spot on the screen and it is only possible to move forward with the TAB command.

The kernal already contains routines for arbitrarily positioning the cursor, however. Two memory locations in page zero are reserved for the row and column of the cursor position. By reading these values with PEEK we can determine the cursor position at any time.

```
100 PRINT "THE CURSOR IS IN LINE"PEEK(214)"COLUMN"PEEK(211)
```

If we want to set the cursor, it is not enough to just POKE the appropriate values in addresses 214 and 211. The kernal must still calculate the required pointer for screen and color RAM based on the cursor position. There is a routine in the kernal that will do this for us.

```
100 REM SET CURSOR
110 INPUT "ROW";R
120 INPUT "COLUMN";C
130 POKE 214,R
140 POKE 211,C
150 SYS 58640
160 PRINT "TEST";
```

Calling 58640 with SYS 58640 sets the cursor at the position determined by locations 214 and 211.

The combination of these two procedures gives us new capabilities for programming. You can provide status lines in your programs, for instance, in which information can be given to the user from time to time. So as not to disturb the rest of the screen, save the current position before moving the cursor to the status line. Then print the message on the status line, set the cursor position back to the original value, and continue with the execution of the program. A program fragment might look like this:

```
300 R=PEEK(214): REM ROW
310 C=PEEK(211): REM COLUMN
320 POKE 214,0: REM CURSOR IN ROW 0
330 POKE 211,10: REM CURSOR IN COLUMN 10
335 SYS 58640
340 PRINT "PLEASE INSERT DISK"
350 POKE 214,R: REM RESTORE ROW
360 POKE 211,C: REM COLUMN
370 SYS 58640
```

The rows are numbered from 0 to 24 and the columns from 0 to 39.

3.2 Turning the cursor on and off

The cursor marking the current screen position on the Commodore 64 is automatically turned on when the computer is expecting input. This is the case when an INPUT command is executed, for example. When you perform input with GET, however, no cursor appears. There are times however when it would be nice to have the cursor flashing when using GET so that the user is aware that the program is expecting input.

The Commodore 64 has a memory location (204) that functions as a flag for the cursor. If this location contains the value 1 (or any other value not equal to zero), the computer knows that the cursor is turned off and a jump is made (during the interrupt) to the corresponding location in the kernal. A value of zero tells the computer to flash the cursor.

We can make use of this fact when we want to turn the cursor on and off under program control. We can, for example, turn the cursor on before a GET command, then wait for a key press and turn the cursor off.

```
100 POKE 204,0 : REM CURSOR ON
110 GET A$: IF A$="" THEN 110: REM WAIT FOR KEY PRESS
120 POKE 204,1: REM CURSOR OFF
130 PRINT A$;
```

It may happen that the cursor is turned on and immediately turned off while it is still in the on phase. If this happens, a white square will remain on the screen. This can be avoided if one first checks to see if the cursor is in the on phase before it is turned off. There is also a memory location in page zero to accomplish this. Inserting the following line into our example program will cause the computer to wait until the cursor is in the off phase before turning it off.

```
115 IF PEEK(207) THEN 115: REM WAIT UNTIL CURSOR IS OFF
```

You can find an application of this technique in section 3.5.

3.3 Repeat function for all keys

You have no doubt noticed while working with your Commodore 64 that the cursor control keys and the space bar repeat when held down. This is especially useful for positioning the cursor and editing programs. With just a simple POKE command, the repeat function can be extended to all keys. This is particularly helpful for such things as word processing. The switch can be made in the direct mode or in a program and can also be switch ed back by either of these methods. The address used to make the switch is 650. A value of 0 means that only the cursor keys are automatically repeated. If you write the value 128 into memory location 650 with POKE, all keys will repeat. It is also possible to turn the repeat function off entirely by placing the value 64 in address 650.

```
100 POKE 650,128: REM REPEAT FOR ALL KEYS

200 POKE 650,0   : REM REPEAT FOR CURSOR ONLY

300 POKE 650,64  : REM TURN REPEAT OFF
```

The repeat delay and repeat rate values are found in locations 651 and 652, respectively. These values are always renewed by the kernal, so changes are only possible by moving the kernal to RAM (see sections 2.6 and 4.2).

3.4 The WAIT command: Waiting for a key press

The WAIT command is a little-used BASIC command. We will show you what it does and what it can be used for.

WAIT A,B

This command gets the contents of memory location A (as in a PEEK command), and ANDs this value with B. If the result is not zero, program execution continues. It is assumed in this description that the value of A is either the address of an I/O port or some other peripheral, or that the value of A is changed by an interrupt. Otherwise the command will either wait forever or not at all.

The most interesting use is waiting for a certain key press. Memory location 653, for example, contains information about whether or not the SHIFT, COMMODORE, or CTRL keys have been pressed. You can use the WAIT command to wait until one of these keys has been pressed.

```
100 PRINT "PRESS THE CONTROL KEY"
110 WAIT 653,4: REM WAIT FOR CTRL
120 ...
```

In line 110 the program waits until the control key is pressed. You can wait for the shift and Commodore keys with the following WAIT commands:

```
WAIT 653,1: REM WAIT FOR SHIFT

WAIT 653,2: REM WAIT FOR COMMODORE KEY
```

If you want to wait for any desired key press, you can check location 203. If no key is pressed, this location contains then value 64, otherwise it contains the matrix number of key pressed (see section 2.6). With

```
WAIT 203,64
```

the program waits as long as a key pressed. With

```
WAIT 203,63
```

the computer waits until a key is pressed. This key can then be determined through use of the GET command, for example.

```
100 WAIT 203,63
110 GET A$: PRINT A$;
```

The WAIT command will be ended only when a key is pressed. If there is data already in the keyboard buffer, you can also make the number of pressed keys the basis of the WAIT command.

```
100 WAIT 198,255
110 GET A$: PRINT A$;
120 GOTO 100
```

3.5 Assigning the function keys

The Commodore 64 has, in addition to its alphanumeric keys, four function keys, each of which has two functions. These keys can be used for menu control, for instance, in order to select a certain part of a program. These keys can be polled with GET and then execution can be transferred depending on the key pressed. The function keys have the following ASCII codes:

```
f1  =>  133
f3  =>  134
f5  =>  135
f7  =>  136
```

Pressing the shift key at the same time increments the ASCII value by four:

```
f2  =>  137
f4  =>  138
f6  =>  139
f8  =>  140
```

The function keys can be polled in the following manner:

```
100 GET A$ : IF A$="" THEN 100
110 A = ASC(A$)
120 IF A = 133 THEN 1100 : REM F1
130 IF A = 134 THEN 1200 : REM F3
140 IF A = 135 THEN 1300 : REM F5
150 IF A = 136 THEN 1400 : REM F7
160 IF A = 137 THEN 1500 : REM F2
```

```
170 IF A = 138 THEN 1600 : REM F4
180 IF A = 139 THEN 1700 : REM F6
190 IF A = 140 THEN 1800 : REM F8
200 GOTO 100
```

Control is passed to the appropriate line based on the function key that was pressed. This can be accomplished in a more elegant fashion with an ON ... GOTO statement.

```
100 GET A$ : IF A$="" THEN 100
110 A = ASC (A$) : IF A<133 OR A>140 THEN 100
120 ON A-132 GOTO 1100,1200,1300,1400,1500,1600,1700,1800
```

This technique can be used within a program. We would now like to present to you a program which allows a string of characters to be assigned to each function key and which will display this string on the screen whenever the function key is pressed. The function keys could be assigned with BASIC command words, for instance. It is also possible to assign a word followed by a RETURN to a function key. This allows the command to be executed directly. If, for example, the string "LIST", followed by the code for RETURN, is assigned to the F1 key, then the program currently in memory will be listed whenever the F1 key is pressed. The maximum length of the string assigned to a function key is 10 characters, the length of the keyboard input buffer.

With our program, you can assign not just eight different strings to the function keys (dual assignment-- with or without the shift key), but sixteen. The Commodore and control keys are used along with the shift key to select the desired function from among the four keys. We have chosen the following assignments for the keys:

f1 => f1
f2 => f3
f3 => f5
f4 => f7
f5 => f1 plus SHIFT
f6 => f3 plus SHIFT
f7 => f5 plus SHIFT
f8 => f7 plus SHIFT
f9 => f1 plus COMMODORE
f10 => f3 plus COMMODORE
f11 => f5 plus COMMODORE
f12 => f7 plus COMMODORE
f13 => f1 plus CTRL
f14 => f3 plus CTRL
f15 => f5 plus CTRL
f16 => f7 plus CTRL

Here is the machine language program which allows the assignment of the function keys. The strings will be placed in memory by a BASIC program.

```
0001 033C                                        ;FUNCTION KEY
S FOR CBM 64
0002 033C                                        ;
0003 033C                                        ;
0004 033C                  ORG 828               ;CASSETTE BUF
FER
0005 033C           KEYVEC EQU $28F              ;VECTOR FOR K
EYBOARD DECODING
0006 033C           KEYPNT EQU $F5               ;POINTER TO D
ECODER TABLE
0007 033C           BUFFER EQU $277              ;KEYBOARD BUF
FER
0008 033C           NOKEYS EQU $C6               ;NUMBER OF CH
ARACTERS IN KEYBOARD BUFFER
0009 033C           SHIFT  EQU $28D              ;FLAG FOR SHI
FT/CBM/CTRL
0010 033C           KEYNO  EQU $CB               ;MATRIX NUMBE
R FOR PRESSED KEY
0011 033C           LSTKEY EQU $C5               ;NUMBER OF LA
ST KEY
0012 033C           TEMP   EQU LSTKEY
```

```
033C             FMIN    EQU $85          ;CODE FOR LOW
EST FUNCTION KEY
033C             FMAX    EQU $88          ;CODE FOR HIG
HEST FUNCTION KEY
033C             SETFLG  EQU $EB26
033C             OLDKEY  EQU $EB48        ;OLD KEYBOARD
 ASSIGNMENT
033C                                      ;
033C A9 47       INIT    LDA #<FNCTN      ;SET VECTOR T
O NEW ROUTINE
033E A0 03               LDY #>FNCTN
0340 8D 8F 02            STA KEYVEC
0343 8C 90 02            STY KEYVEC+1
0346 60                  RTS
0347                                      ;
0347 A4 CB       FNCTN   LDY KEYNO        ;KEY NUMBER
0349 C4 C5               CPY LSTKEY       ;SAME AS BEFO
RE?
034B F0 0A               BEQ NOFUNC       ;YES
034D B1 F5               LDA (KEYPNT),Y   ;ASCII CODE
034F C9 89               CMP #FMAX+1      ;COMPARE WITH
 HIGHER FUNCTION KEY
0351 B0 04               BCS NOFUNC       ;NO FUNCTION
KEY?
0353 C9 85               CMP #FMIN
0355 B0 03               BCS FTN
0357 4C 48 EB    NOFUNC  JMP OLDKEY       ;TO OLD KEYBO
ARD EVALUATER
035A E9 85       FTN     SBC #FMIN
035C 85 C5               STA TEMP
035E 0A                  ASL
035F 0A                  ASL              ;TIMES 10
0360 65 C5               ADC TEMP
0362 0A                  ASL
0363 AE 8D 02            LDX SHIFT        ;FLAG SHIFT/C
BM/CTRL
0366 E0 01               CPX #1           ;SHIFT?
0368 F0 0E               BEQ SHIFTK
036A E0 02               CPX #2           ;CBM?
036C F0 07               BEQ CBMKEY
036E E0 04               CPX #4           ;CTRL?
0370 D0 09               BNE NOSPEC
0372 18          CTRLKY  CLC
0373 69 28               ADC #40
0375 18          CBMKEY  CLC
0376 69 28               ADC #40          ;POINTER TO N
EXT TABLE
0378 18          SHIFTK  CLC
0379 69 28               ADC #40
037B AA          NOSPEC  TAX              ;POINTER TO I
NDEX
037C A0 00               LDY #0
037E BD 00 CF    GETKEY  LDA TABLE,X      ;GET ASSIGNME
NT FROM TABLE
0381 F0 09               BEQ ENDKEY
```

```
0056 0383 99 77 02          STA BUFFER,Y          ;AND WRITE IN
 BUFFER
0057 0386 E8                INX
0058 0387 C8                INY
0059 0388 C0 0A             CPY #10               ;10 CHARACTER
 ALREADY
0060 038A D0 F2             BNE GETKEY
0061 038C 84 C6      ENDKEY STY NOKEYS            ;SAVE NUMBER
OF CHARACTERS
0062 038E A2 FF             LDX #$FF              ;FLAG FOR INV
ALID KEYBOARD CODE
0063 0390 4C 26 EB          JMP SETFLG            ;ACTUALIZE FL
AGS
0064 0393            TABLE  EQU $CF00
```

The following BASIC program generates the machine language code and places the strings for the 16 function keys into memory. The strings themselves are stored in the program in DATA statements at line 300 and can naturally be changed as desired. Remember that the strings may not be more than ten characters long. If you want to execute a command immediately upon pressing the function key, a RETURN character must terminate the string. This can be done by placing a left-arrow as the last character in the string. When loading the strings into memory, this character will be converted to a RETURN (line 250). If you want to use a quotation mark within the string, use an apostrophe instead (see line 260). If a comma appears in the string, the string must be enclosed in quotation marks (as in line 330).

```
100 FOR I = 828 TO 914
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 169, 71,160,  3,141,143,  2,140,144,  2, 96,164
130 DATA 203,196,197,240, 10,177,245,201,137,176,  4,201
140 DATA 133,176,  3, 76, 72,235,233,133,133,197, 10, 10
150 DATA 101,197, 10,174,141,  2,224,  1,240, 14,224,  2
160 DATA 240,  7,224,  4,208,  9, 24,105, 40, 24,105, 40
170 DATA  24,105, 40,170,160,  0,189,  0,207,240,  9,153
180 DATA 119,  2,232,200,192, 10,208,242,132,198,162,255
190 DATA  76, 38,235
200 IF S <> 10591 THEN PRINT "ERROR IN DATA!!" : END
210 PRINT "OK"
```

```
215 SYS 828
220 REM PLACE KEY ASSIGNMENTS IN MEMORY
230 AD = 12*4096+15*256 : REM $CF00
240 FOR I=0 TO 15 : READ X$ : FOR J=1 TO LEN(X$)
250 A=ASC(MID$(X$,J,1)) : IF A=95 THEN A=13 : REM RETURN
260 IF A=39 THEN A=34 : REM QUOTE
270 POKE AD+10*I+J-1,A : NEXT
280 IF J<>11 THEN POKE AD+I*10+J-1,0 : REM END CRITERIUM
290 NEXT
300 DATA LIST←,RUN←,GOTO,CHR$(
310 DATA ?FRE(0)←,SAVE,PRINT,THEN
320 DATA POKE,PEEK(,PRINT#,INPUT#
330 DATA "LOAD'$',8←",NEXT,GOSUB,RETURN
```

The strings assigned to the function keys will be placed in free RAM at address $CF00. If you are using this memory area or you want to store the strings somewhere else, you must change the address in line 230 as well as the address in the DATA statement in line 170. Replace the fourth and fifth-last elements (0 and 207) with the low and high bytes of the new address. When you use a different address, remember that at least 160 bytes must be available there (16 keys * 10 characters).

To change the function key assignments, all that is required is to change the strings in the DATA statements starting at line 220.

Pressing RUN/STOP-RESTORE will unassign the function keys. You can restore them with SYS 828.

3.6 An Easy INPUT Routine

You have no doubt run across the problem of having your program "interrupted" after invalid input from the keyboard.

There are two primary reasons for this:

Input in the form INPUT A

A program interruption occurs if the input does not consist of exclusively numeric characters.

Input in the form INPUT A$

The program may crash if the RETURN key is pressed without previous alphanumeric input or if too few characters are entered.

Input by means of GET A$ eliminates the first problem and can be used to eliminate the second, but many avoid it because of the necessity of building a string one character at a time if the input is longer than one character. In addition, no flashing cursor is displayed, something that would be desirable as a request for input.

Once you have made sure that the obstacles are removed from data entry, it is still possible that data errors may creep in, which, while they will not cause the program to crash, may result in an erroneous result.

Let us look at a typical example using INPUT A$ to input numeric data.

You want characters from the keyboard which you intend to convert to a numerical value by means of VAL(A$). You have avoided possible conflicts here that would have occurred with INPUT A (entering alphabetic data), although an illegal input has the following effect:

You answer the INPUT with 123R56. The conversion with A=VAL(A$) puts the value 123 in A, certainly not the number intended.

Perhaps you object, saying that such input errors are the exception and not the rule and that getting the wrong answer now and then is not of much consequence in personal computing.

We are of the opinion, however, that it should be your goal to produce "bomb-proof" programs, taking into consideration that you may have the opportunity to write programs which are not just for your own use.

We want to present you with a ready-to-use subroutine which virtually eliminates the problems mentioned earlier. We describe certain parts of this program in detail which you can adapt to your own needs.

First, the meanings of the variables and memory locations used in the program are explained. The following variables must be initialized before the subroutine is called:

MN=0	Purely numerical input is desired.
MN=1	The input may be alphanumeric.
ML=0	The length given in IL is mandatory.
ML=1	The length given in IL is the maximum.
IL	Mandatory or maximum length of the input

Furthermore, the routine uses the following variables:

CC	Number of valid characters in IN$
CS	Current cursor column
CZ	Current cursor line
CP	Length created by inserting an input field
MS	Highest cursor column during input

G$ Contains the character from the last GET
IN$ The complete input is returned in this variable

Memory locations used:
204=0 Turn cursor on
204=1 Turn cursor off
205 Counter for the flash frequency of the cursor
207=0 Cursor in OFF phase
207=1 Cursor in ON phase
211 Cursor column
214 Cursor line

One additional preparation necessary for using this routine is opening the screen with OPEN 1,3. This is required because the created input is read from the screen by means of GET#1 in line 35680.

Now the individual program steps:

35020 The variables are initialized and the cursor position is saved for the GET#1 in line 35680.
35060 A character is read from the keyboard.
35080 If this character is a RETURN, the input is ended depending on the length and the value in ML.
35100-35130 If the DELETE key was used, the length and position counters are actualized if the input field contains only legal characters (CP=0) or if it was enlarged with INSERT.
35140 INSERT is only executed if the length in IL will not be exceeded.
35160-35180 Ensures that the cursor does not leave the input field when CRSR RIGHT and LEFT are used.

35200 Entry point of the data filter depending on MN.

35220-35240 If the cursor is within the data field and a purely numerical value will be entered, the characters will be accepted. The legal range, here set at values 47 through 58, can be changed as desired.

In our example these values form the borders for the representable digits 0-9. You can find the appropriate values for the border of your choosing on page 135ff of the Commodore 64's user's guide.

For our example that means that 47 corresponds with the character 0 and 58 stands for the character 9. All characters in between (0-9) are also legal.

35300-35380 Here the same thing happens, but the legal range is expanded to include the letters of the alphabet.

35400 If the input is not long enough (and ML=0), input will not be ended.

35600-35690 The input field is taken from the screen and put into IN$ until either the length given in IL is reached or no more data is found on the screen. Before this can happen, the cursor is reset to the position it had at the beginning of the routine so that the GET#1 starts at the beginning of the field.

36000-36060 The cursor is turned off and the character in G$ is displayed on the screen.

The line numbering of the routine was chosen arbitrarily. The routine may start with 1000, 50000, or some other number of your own choosing. Remember, however, to

change the line number references in GOTO, GOSUB, and IF...THEN statements accordingly.

We recommend that you start the subroutines of each of your programs with the same line number. This makes it easier to write new programs using this subroutine library.

Here now is the INPUT routine:

```
35000 REM  INPUT FROM KEYBOARD
35020 IN$="":
      CC=0:
      CS=PEEK(211):
      CZ=PEEK(214):
      CP=0:
      MS=0
35040 POKE 204,0:
      REM  CURSOR ON
35060 GET G$:
      IF G$=""
      THEN 35060
35080 G=ASC(G$):
      IF G=13
      THEN ON ML+1
        GOTO 35400,35600
35100 IF G=20 AND CP>0
      THEN CP=CP-1:
        GOSUB 36000:
        GOTO 35060:
        REM  DELETE
35120 IF G=20 AND CC>0 AND PEEK(211)>CS
      THEN CC=CC-1:
        MS=MS-1:
        CP=CP-1:
        GOSUB 36000:
        GOTO 35060
35130 IF G=20 AND PEEK(211)>CS
      THEN MS=MS-1:
        GOSUB 36000:
        GOTO 35060
35140 IF G=148 AND CP+MS<IL
      THEN CP=CP+1:
        MS=MS+1:
        GOSUB 36000:
        GOTO 35060:
        REM  INSERT
35160 IF G=29 AND PEEK(211)<=CS+IL-1
      THEN GOSUB 36000:
        GOTO 35060:
        REM  CURSOR RIGHT
```

```
35180 IF G=157 AND PEEK(211)>CS
      THEN GOSUB 36000:
        GOTO 35060:
        REM  CURSOR LEFT
35200 ON MN
      GOTO 35300
35220 IF G>47 AND G<58 AND CC<IL AND PEEK(211)<=CS+IL-1
      THEN CC=CC+1:
        GOSUB 36000:
        GOTO 35360
35230 GOTO 35360
35240 IF G>47 AND G<58 AND PEEK(211)<=CS+IL
      THEN GOSUB 36000:
        GOTO 35360
35300 IF G<48 OR (G>57 AND G<65) OR (G>90 AND G<193) OR
      G>218
      THEN 35060
35320 IF CC<IL AND PEEK(211)<=CS+IL-1
      THEN CC=CC+1:
        GOSUB 36000:
        GOTO 35360
35340 IF PEEK(211)<CS+IL
      THEN GOSUB 36000
35360 IF CP>0
      THEN CP=CP-1
35380 GOTO 35060
35400 IF CC<>IL
      THEN 35060
35600 POKE 205,2
35620 IF PEEK(207)<>0
      THEN 35620
35640 POKE 204,1
35660 POKE 211,CS:
      POKE 214,CZ
35670 IF CC=0
      THEN RETURN

35680 GET #1,G$:
      IF G$=CHR$(13)
      THEN IN$=LEFT$(IN$+"                                ",IL):
        RETURN

35682 IN$=IN$+G$
35684 IF LEN(IN$)<IL
      THEN 35680
35690 RETURN

36000 POKE 205,2
36020 IF PEEK(207)<>0
      THEN 36020
36040 PRINT G$;:
      IF PEEK(211)>MS
      THEN MS=PEEK(211)-CS
36060 RETURN
```

How do you use this program?

Suppose you want to enter an item number for an inventory. This number must be exactly six digits long and consist of numeric characters only.
We would program the following to accomplish this:

```
10 OPEN 1,3
100 IL=6:MN=0:ML=0
110 PRINT"PART NUMBER ";:GOSUB35000
120 IN=VAL(IN$)
```

The desired part number is now at your disposal in IN and you can be sure that is exactly six digits long and that it contains only numerical digits.

Along with the previously entered item number you also need the description of the part. Since you have set up a file with records of a predetermined length, this description may be no longer than a certain value, say 10 characters. This is the maximum length; it is not obligatory.

The appropriate program lines look like this:

```
200 IL=10:MN=1:ML=1
210 PRINT"DESCRIPTION ";:GOSUB35000
```

The description is now contained in IN$ and, if less than 10 characters long, padded with blanks at the end.

The price is also important of course. It has a variable length, up to, say, eight characters and is strictly numeric.

```
300 IL=8:MN=0:ML=1
310 PRINT"PRICE ";:GOSUB35000
320 IN=VAL(IN$)
```

The number, consisting of a maximum of eight digits, is now in IN and you can proceed with the input of the quantity and so on.

We hope that this small routine takes care of the problems you may have had with syntactically incorrect data input. Feel free to make use of the special features used in the subroutine in regard to the cursor positioning and the input from the screen (GET#1) in your own programs.

3.7 A "mouse" for the CBM 64

A new buzz word has infiltrated the world of personal computers: the "MOUSE"

What is behind this intriguing expression?

You are probably acquainted with devices called track balls on video games. A track ball is a pointing or control device used instead of joysticks or paddles to move figures around on the screen.

In contrast to joysticks whose handles can be moved in one of only eight directions, the track ball allows rotation in all directions since it employs a free-moving ball without axes, whose movement is converted into two angles for the X and Y axes. On video games this ball is operated with the palm.

The "mouse" consists of such a ball built into the underside of a housing about the size of a package of cigarettes, which one lays, ball down, on the table and rolls back and forth with the hand.

Through the movement of this box on the table the ball is in turn set in motion by the friction against the table surface. The coordinates of the device are transmitted to the computer via a special interface.

How does one make serious use of a mouse?

If a program intended for a large range of users is supposed to bear the title "user-friendly," it will in all probability be designed using what is called the menu technique. This procedure has the advantage that it can easily be understood and used by almost anyone. The user can select the desired function either directly or through a succession of choices, each more specific than the last and all presented to him on the screen.

The choice is made either by entering a number or letter corresponding to an option on the screen or by moving the cursor to the desired point on the screen.

Experts in ergonomics have discovered that the operations involved in making a choice can be accomplished more comfortably and more certainly when one does not have to search for the appropriate key on the keyboard but rather when one is resting comfortably in an easy chair. With the mouse, the movements of the cursor on the screen correspond directly to those of the device on the table. Reaching the desired field is signaled by pressing a button on top of the mouse.

In order to give you the opportunity to experiment with this charming little animal without requiring the purchase of an expensive track ball, we have developed the following program which works with the conventional joystick in control port 2 of your Commodore 64.

Naturally, this will not allow the same ease of use as the real mouse, but the experimentation with the principle of the thing will answer for our purpose.

In keeping with our usual style, we first present the variables and memory locations used and then discuss the program in detail.

First the variables:

RO$	character for reverse on
RF$	character for reverse off
A$	character entered
B$	after RETURN contains all the previously entered characters
A	two-dimensional variable field which contains the ASCII values for each of the characters on the first four lines of the video display.
DR	original value of the data direction register in 56322. This value must be POKEd back into this location at the end of the program.
J	position of the joystick in control port 2
JS	joystick column
JZ	joystick line
PS	column position for PRINT
PZ	line position for PRINT
S	column of the joystick cursor for indexing of A(X,Y)
Z	as above, but line

Memory locations:

56322	data direction register for control port 2
58643	kernal routine for determining cursor position
58636	kernal routine for positioning the cursor
781	contents of processor register X, loaded from here by the SYS command and placed back when the routine ends
782	as above, but for the Y register
204	=0 cursor on
	=1 cursor off
205	counter for flash frequency of cursor
207	=0 cursor in OFF phase

<>0 cursor in ON phase

Step-by-step description of the program:

1	Because the control ports and the keyboard use the same peripheral interfaces in the C64, the keyboard is turned off here. At the end of your program the value in DR must be poked back into 56322 or the computer will not respond to the keyboard. Only STOP/RESTORE will get you out of our example program.
10-50	The menu field is constructed on the screen.
60-560	The array is filled with the ASCII values of the characters in the first four lines of the display. When this array is indexed by the line and column positions, it returns the value of the character at that screen position.
680	The character produced by the subroutine at 5000 is displayed on the screen.
700	The cursor position resulting from the PRINT is saved because the menu field will be reconstructed in line 720. This is necessary because the lines scroll up when the screen is full and the field may be destroyed.
760	The cursor, displaced by the reconstruction of the field, is returned to its original position.
780	If the last character was RETURN, the input of a line is terminated. If you wish to perform further operations on the data, you should take the data out of B$,
800	otherwise the entered character is appended to B$.

5020-5140 The cursor is saved and turned off, set to position 0, and turned on again.

5160 The value obtained form the joystick on control port 2 is put into J.

5170 Delay loop--makes the cursor easier to control.

5180-5340 The cursor is moved according to the position of the joystick.
If the joystick button was pressed (5260), the character under the cursor (at 6010) is put into A$.

6010-6160 The array A(X,Y) is addressed with the cursor position of the joystick and the resulting value is placed in A$ (6060).
Because the C64 has a double-line organization, that is, the column counter can go up to 80 positions although the screen is only 40 columns wide, the column value is corrected for properly indexing the array in line 6050.

This program is quite simple to use. After typing RUN, the cursor appears in the upper left-hand corner of the display. You can move it about with a joystick plugged into control port 2. When you come to a character that you would like to put in B$, simply press the fire button on the joystick to do so. As acknowledgment, the chosen character appears several lines down. Now you can go on to the next character.

After selecting RETURN, the line is complete in B$ and you can process it as desired.

We hope that you have fun with this program and that it encourages you to try similar ideas of your own.

The program listing:

```
1     DR=PEEK(56322):
       POKE 56322,224:
       RO$=CHR$(18):
       RF$=CHR$(146)
5     PRINT CHR$(147):
       GOSUB 10:
       GOTO 60
10    PRINT CHR$(19)" , - . / 0 1 2 3 4 5 6 7 8 9
          ";
20    PRINT "  @ A B C D E F G H I J K L            ";
30    PRINT " M N O P Q R S T U V W X Y Z           ";
40    PRINT " "RO$"RET"RF$" "RO$"DEL"RF$" "RO$"F1"RF$" "R
       O$"F3"RF$" "RO$"F5"RF$;
45    PRINT " "RO$"F7"RF$"                  "
50    RETURN

60    DIM A(4,40)
100   FOR I=0 TO 13
120     A(0,I*2+1)=I+44
140   NEXT I
180   FOR I=0 TO 12
200     A(1,I*2+2)=I+64
220   NEXT I
260   FOR I=0 TO 13
280     A(2,I*2+1)=I+77
300   NEXT I
340   FOR I=1 TO 3
360     A(3,I)=13
380   NEXT I
420   FOR I=5 TO 7
440     A(3,I)=20
460   NEXT I
500   FOR I=0 TO 3
520     A(3,I*2+9)=I+133
560   NEXT I
580   PRINT :
       PRINT
600   B$="":
       X=FRE(0)
640   GOSUB 5000:
       REM  GET CHARACTER
680   PRINT A$;
700   SYS 58643:
       PZ=PEEK(211):
       PS=PEEK(214)
720   GOSUB 10
760   POKE 211,PZ:
       POKE 214,PS:
       REM  SYS58636
```

```
780  IF ASC(A$)=13
      THEN 600
800  B$=B$+A$
820  GOTO 640
5000 REM
5001 REM  ***** READ JOYSTICK *****
5002 REM
5020 SYS 58643:
      REM  SAVE PRINT-CURSOR
5060 PZ=PEEK(781):
      PS=PEEK(782)
5070 POKE 205,3
5080 IF PEEK(207)
      THEN 5080
5090 POKE 204,1
5100 POKE 781,Z:
      POKE 782,S:
      JZ=Z:
      JS=S
5120 SYS 58636:
      REM  SET JOYSTICK CURSOR
5140 POKE 204,0:
      REM  TURN CURSOR ON
5160 J=PEEK(56320):
      REM  READ JOYSTICK
5170 FOR I=0 TO 30:
      NEXT I
5180 IF (J AND 1)=0
      THEN JZ=JZ-1
5200 IF (J AND 2)=0
      THEN JZ=JZ+1
5220 IF (J/4)=0
      THEN JS=JS-1
5240 IF (J AND 8)=0
      THEN JS=JS+1
5260 IF (J AND 16)=0
      THEN 6000
5280 IF JZ<0
      THEN JZ=0
5281 IF JS<0
      THEN JS=0
5282 IF JS>30
      THEN JS=30
5283 IF JZ>3
      THEN JZ=3
5285 POKE 205,3
5290 IF PEEK(207)
      THEN 5290
5295 POKE 204,1
5300 POKE 781,JZ:
      POKE 782,JS:
      SYS 58636
5340 GOTO 5140
```

```
6000 REM
6001 REM  ***** GET ascii VALUE OF CHARACTER *****
6002 REM
6010 POKE 205,3
6015 IF PEEK(207)
      THEN 6015
6017 POKE 204,1
6020 SYS 58643:
      REM  GET CURSOR POSITION
6040 Z=PEEK(781):
      S=PEEK(782)
6050 IF S>39
      THEN S=S-40
6060 A$=CHR$(A(Z,S))
6100 POKE 781,PZ:
      POKE 782,PS
6120 SYS 58636:
      REM  LOAD PRINT POSITION
6160 RETURN
```

Chapter 4 Advanced BASIC

4.1 Creating a BASIC line in BASIC

Have you ever tried to write a universal computer program? By universal we mean a program which can be executed with any desired arithmetic operations with any variables or constants."

Of course not, you will answer. The operation of a program depends on the previously entered algorithms. This is true, but imagine for a moment that you want to write a word processor that allows calculations. Within the text are numeric fields on which the mathematical operations are to be performed. Such a program might combine the features of a word processor with those of a spread-sheet program.

You could write a version of this program that was set up to do only certain calculations, such as balancing a checkbook or something similar. It could not perform general calculations for which it was not specifically designed, however. To perform other operations, a new version of the program would have to be written. It would be more practical to have a version which would allow any calculations to be performed.

This is what we want to present to you, a procedure which allows you to specify the variables on which arithmetic operations are to be performed and what operations are to be executed while the program is RUNning.

This is only possible if we can generate a BASIC line containing the desired formula within the executing program. We will show you how this can be done.

The following program contains a machine language subroutine, but this will be handled entirely from BASIC. Before we discuss the individual program steps, we first present the variables and memory addresses used.

First the variables:

TM	Contains the last address in memory
VL	Least-significant byte of the address "variable start"
VH	As above, but most-significant byte
VT	As above, but total value
BU	Address of the line input buffer
BC	Index variable for filling the buffer
CA$	Variable containing the calculation
RE	Contains the result after executing the routine

The memory locations used:

45-46	Pointer to the start of the variable table
47-48	Pointer to the start of the arrays
49-50	Pointer to the start of the strings
56	Most-significant byte of the pointer to the end of BASIC RAM
40448	The created BASIC line 50100 is placed here.
40704	Address of the routine which creates a BASIC line from the contents of the input buffer and puts it in 40448

Step-by-step description of the program operation:

1 The top-of-memory is set to 40448. Memory above this point will be used for the machine language routine and later for the created BASIC line.

2-6 The pointer for the start of the variable table is raised because the connecting line 50099 will be inserted here.

10-14 Lines 50100 and 50110 are established so that these are available at all times in case a jump is made to them without having previously placed an operation there. The lines contain PRINT and RETURN.

20-30 The connecting line 50099 is placed at the end of the BASIC program. The continuation address of this line points to the line 50100 at address 40449.

32-50 These lines contain the machine language program which will be examined in greater detail later.

60-70 The machine language program is placed in memory at 40704.

50040 The BASIC line is read in from the keyboard into CA$. Make sure that only functions are entered.

50050 The input is taken from CA$ and placed in the line input buffer (up to 50075).

50080 The machine language program to create the line is called.

50095 Here the created calculation is called. The result will be returned in RE.

For those who are interested, here is the machine language program:

```
LDA $7A        save BASIC pointer
STA $9FFF
LDA $7B
STA $9FFE
LDA $14
STA $9FFD
LDA $15
```

```
        STA $9FFC
        LDA #$0B        offset to input buffer
        STA $7A
        JSR $A579       call the routine "CRUNCH"
        LDX #0
XX      LDA $0200,X     transfer line to 40453
        BEQ YY          jump out when done
        STA $9E05,X
        INX
        BNE XX
YY      LDA #$3A        behind the line
        STA $9E05,X
        LDA #$8E        append a RETURN
        STA $9E06,X
        LDA #0          mark the end-of-line
        STA $9E07,X
        STA $9E08,X
        STA $9E09,X
        LDA $9FFF       reload BASIC pointer
        STA $7A
        LDA $9FFE
        STA $7B
        LDA $9FFD
        STA $14
        LDA $9FFC
        STA $15
        RTS             back to BASIC
```

The program listed below consists of two parts:

The first part from line 1 to 70 need be executed only once, at the beginning of the program. It is important that these lines also be used at the beginning of your program

and not be moved to other line numbers, otherwise your variables will be destroyed in lines 1 and 6.

The second part of the program makes a BASIC line out of the formula entered in CA$ and executes this. The result is returned in RE.

The line numbers of the program in which these routines are placed may not exceed 49999. The lines at 50000 must absolutely be the last in the program.

The only restriction when using these routines is that you must only enter functions, though these may be of any type, such as 75/2*V1-V2+SQR(V3). The assignment of the result to RE is already done in line 50050.

WARNING! Once the program has been started, it may not be changed. You should enter NEW, reload the program, and then change it. This is necessary because the created lines are not placed directly behind the BASIC program but high in memory. As a result, the computer may crash if you try to edit, insert, or delete a line.

Here is the program listing:

```
1       POKE 56,158:
        CLR
2       IF PEEK(45)+2>255
        THEN POKE 45,2-(256-PEEK(45)):
          POKE 46,PEEK(46)+1:
          GOTO 6
4       POKE 45,PEEK(45)+2
6       POKE 47,PEEK(45):
        POKE 48,PEEK(46):
        POKE 49,PEEK(45):
        POKE (50),PEEK(46)
8       TM=40448
10      POKE TM,0:
        POKE TM+1,7:
        POKE TM+2,158:
        POKE TM+3,180:
        POKE TM+4,195
```

```
12     POKE TM+5,153:
       POKE TM+6,0:
       POKE TM+7,13:
       POKE TM+8,158:
       POKE TM+9,190
14     POKE TM+10,195:
       POKE TM+11,142:
       POKE TM+12,0:
       POKE TM+13,0:
       POKE TM+14,0
20     VL=PEEK(45):
       VH=PEEK(46):
       VT=VH*256+VL
30     POKE VT-4,1:
       POKE VT-3,158:
       POKE VT-2,179:
       POKE VT-1,195
32     DATA 165,122,141,255,159,165,123,141,254,159,165,2
       0,141,253,159,165,21
33     DATA 141,252,159,169,11,133,122,32,121,165
34     DATA 162,0,189,0,2,240,6,157,5,158,232,208,245,169
       ,58,157,5,158
36     DATA 169,142,157,6,158,169,0,157,7,158,157,8,158,1
       57,9,158
40     DATA 173,255,159,133,122,173,254,159
50     DATA 133,123,173,253,159,133,20,173,252,159,133,21
       ,96
60     FOR I=40704 TO 40785
70       READ MC:
         POKE I,MC:
       NEXT I
50000 REM    CALCULATOR *************
50040 BU=523:
      INPUT "CALC";CA$
50050 POKE BU,ASC("R"):
      POKE BU+1,ASC("E"):
      POKE BU+2,ASC("="):
      BU=BU+2
50060 FOR I=1 TO LEN(CA$)
50070   POKE BU+I,ASC(MID$(CA$,I,1)):
      NEXT I
50073 BC=LEN(CA$)+1
50075 POKE BU+BC,0:
      POKE BU+BC+1,0:
      POKE BU+BC+2,0
50080 SYS 40704
50095 GOSUB 50100
50097 RETURN
```

In closing, we have one more suggestion which we would like to present to you.

Assume for one moment that you have found a procedure which will create the BASIC program line necessary to solve a specially formulated problem. Furthermore, this procedure is so universal that from a set of tasks it will create a corresponding set of program lines, including loops and jumps.

The only remaining problem is to place all of these lines in memory one after the other. The procedure we have described in this section can create only a single program line, but it can be expanded so that it can be used for several lines.

It should be noted that the machine language program does not always transfer the created line to the same point in memory but to a address depending on the length of the previously created line. In addition, the continuation pointer (the first two bytes at the beginning of the line) must be taken care of, something we omitted in our example because the return command was placed directly within the created line.

Perhaps you will use this suggestion to write a truly universal program generator, since such a thing is possible in principle. Program generators are programs which are given a specific task of a particular kind and then create a program in a given programming language (it need not be BASIC).

4.2 Copying the BASIC interpreter into RAM

One of the advantages that the Commodore 64 has over the other Commodore computers is that the entire address space of the processor--64 kilobytes--is equipped with RAM. This presents us with some interesting possibilities such as providing the 64 with a completely new operating system and a new BASIC interpreter. You need only load the new or modified kernal or BASIC interpreter into RAM and then tell the computer to switch off the ROM and activate the corresponding RAM. This can be accomplished with POKE commands.

If you do not want to load an entirely new BASIC but only wish to change certain characteristics, such as implementing your own functions or modifications to existing functions or commands, you would simply copy the BASIC interpreter into RAM, execute the modifications there and then switch to RAM.

A short discussion of the Commodore 64's memory management will help explain this process. When the computer is turned on, the kernal ROM and the BASIC ROM are switched on and executed. When you read from this area of memory with a PEEK command, you receive the value from the ROM (see section 9.5 for information on how to read the RAM). If you write to this area with POKE, you will always write to RAM, regardless of whether it is selected or not. We can make use of this feature to copy the entire kernal or BASIC ROM into the underlying RAM in order to manipulate it for our purposes. The copying can be done with a BASIC loop.

```
FOR I=B TO E : POKE I, PEEK(I) : NEXT
```

B is the beginning address and E the end address. For BASIC these addresses are 40960 ($A000) and 49151 ($BFFF); the kernal lies from 57344 ($E000) to 65535 ($FFFF).

This POKE loop copies the contents of the ROM into the underlying RAM. BASIC continues to run in ROM, however. We must tell the computer to switch over to the RAM.

Memory location 1, the processor port, is used for this purpose. Normally this location contains the value 55. If you want to run BASIC in RAM, you must select the RAM with POKE 1,54. Note: You may only execute this POKE after you have copied BASIC from 40960 to 49151 into RAM or the computer will "crash." If you also want to copy the kernal to RAM, this must be done together with the BASIC ROM because the selection of this RAM automatically selects the RAM under the BASIC ROM as well (see section 2.6). The POKE command to make this switch is POKE 1,53. If you want to manipulate the BASIC interpreter, first copy the ROM, make the desired changes, and then save the program with which you made the changes and switch over with POKE 1,.... If you have made an error your computer may "hang up" and you must start over from the beginning. Reload your program, correct the error and run it again until it works as desired.

4.3 No more negative numbers with the FRE function

Have you ever found it surprising that when turn on your Commodore 64 it announces that it has 38911 bytes free but when you issue the command

```
PRINT FRE(0)
```

it responds with -26627?

If one receives a negative number, one must add 2 to the 16th power (or 65536) to the value in order to get the proper (positive) value. This is not overly difficult but it is inconvenient. What is the cause of this?

We must examine the corresponding locations in the ROM listing to determine the answer (address $B37D: The Anatomy of the Commodore 64). There, after the strings which are no longer needed are removed and their memory locations made free (garbage collection), the free memory area is calculated: The start of the strings ($33/$34) minus the end of the arrays ($31/$32). This 16-bit integer is converted into floating-point format and returned. Here is the error. The integer value is treated as a signed number just like the integer variables (%), which can only contain values from -32768 to 32767. If these numbers were treated as positive values, they could contain values in the range 0 to 65535. With the earlier Commodore computers there was never more than 32767 bytes of memory free so that this error was never encountered. We must therefore change the FRE routine so that the conversion to a floating-point number treats the integer as a positive value. This is the case with line numbers which may also be larger than 32767.

These are the changes which are necessary. Here we have placed the additional code in an unused area of the BASIC interpreter.

```
B38D    4C 55 BF    JMP $BF55
B390    EA          NOP

BF55    A5 34       LDA $34
BF57    E5 32       SBC $32
BF59    A2 00       LDX #$00
BF5B    86 0D       STX $0D
BF5D    85 62       STA $62
BF5F    84 63       STY $63
BF61    A2 90       LDX #$90
BF63    4C 49 BC    JMP $BC49
```

The changes can be made with a small POKE loop.

```
100 FOR I=40960 TO 49151
110 POKE I, PEEK(I) : NEXT
120 A=11*4096+3*256+8*16+13
130 FOR I=A TO A+3
140 READ X : POKE I,X : NEXT
150 A=11*4096+15*256+5*16+5
160 FOR I=A TO A+16
170 READ X : POKE I,X : NEXT
180 POKE 1,54
200 DATA 76,85,191,234
210 DATA 165,52,229,50,162,0,134,13,133,98,132,99
220 DATA 162,144,76,73,188
```

4.4 Returning to a BASIC program after a LIST command

When one puts a LIST command within a BASIC program, execution always returns to the command mode after the LIST command is carried out. This is inconvenient when you want to output certain lines such as those which contain function definitions using DEF FN. You are also prohibited from outputting more than one copy of a program listing, even from a loop in the direct mode, such as:

```
FOR I=1 TO 2 : LIST : NEXT
```

The remedy to this problem is as follows: Place a jump to the BASIC warm-start at the end of the LIST function by means of a return command. In addition, the pointer to the program text must be saved before calling the LIST function because this will be changed during the LIST.

We need a small routine which carries out these tasks and jumps back to the BASIC interpreter. Since this requires a machine language program in any case, we will include the code to copy the BASIC ROM into RAM. This way we avoid the slow BASIC POKE loop.

We have placed this routine in the cassette buffer. After entering or loading, it is executed with SYS 828 and immediately allows the use of the LIST command without program interruption.

```
033C               ORG 828        ;CASSETTE BUF
FER
033C        CHRPTR EQU $7A        ;PROGRAM POIN
TER
033C        CHRGOT EQU $79        ;GET LAST CHA
RACTER
033C        LIST   EQU $A69C      ;LIST ROUTINE
033C        LSTVEC EQU $A042      ;POINTER TO L
IST ROUTINE
033C        NEXTST EQU $A8F8      ;SET PROGRAM
POINTER
033C                              ;TO NEXT STAT
EMENT
033C        CRLF   EQU $AAD7      ;OUTPUT CR
033C A2 20         LDX #32        ;COPY 32 PAGE
S
033E A9 A0         LDA #$A0       ;POINTER TO S
TART OF BASIC
0340 A0 00         LDY #0
0342 84 22         STY $22
0344 85 23         STA $23
0346 B1 22  LOOP   LDA ($22),Y    ;TRANSFER LOO
P
0348 91 22         STA ($22),Y
034A C8            INY
034B D0 F9         BNE LOOP
034D E6 23         INC $23
034F CA            DEX
0350 D0 F4         BNE LOOP
0352 A9 60         LDA #$60       ;RTS CODE
0354 8D 14 A7      STA $A714
0357 A9 EA         LDA #$EA       ;NOP CODE
0359 8D BB A6      STA $A6BB
035C 8D BC A6      STA $A6BC
035F A9 6D         LDA #$6D       ;POINTER TO N
EW LIST FUNCTION
0361 8D 42 A0      STA LSTVEC
0364 A9 03         LDA #$03
0366 8D 43 A0      STA LSTVEC+1
0369 A9 36         LDA #$36       ;SWITCH TO RA
M
036B 85 01         STA 1
036D 60            RTS
036E A5 7A  NEWLST LDA CHRPTR
0370 48            PHA            ;SAVE PROGRAM
 POINTER
0371 A5 7B         LDA CHRPTR+1
0373 48            PHA
0374 20 79 00      JSR CHRGOT     ;GET LAST CHA
RACTER
0377 20 9C A6      JSR LIST       ;EXECUTE LIST
037A 20 D7 AA      JSR CRLF       ;OUTPUT CR
037D 68            PLA
037E 85 7B         STA CHRPTR+1
0380 68            PLA            ;GET PROGRAM
POINTER BACK
```

```
0043 0381 85 7A              STA CHRPTR
0044 0383 20 F8 A8           JSR NEXTST          ;POINTER TO N
EXT STATEMENT
0045 0386 4C 79 00           JMP CHRGOT          ;GET LAST CHA
RACTER
```

Here is the loader program in BASIC.

```
100  FOR I=828 TO 904
110    READ X:
         POKE I,X:
         S=S+X:
       NEXT
120  DATA 162,32,169,160,160,0,132,34,133,35,177,34
130  DATA 145,34,200,208,249,230,35,202,208,244,169,96
140  DATA 141,20,167,169,234,141,187,166,141,188,166,169
150  DATA 109,141,66,160,169,3,141,67,160,169,54,133
160  DATA 1,96,165,122,72,165,123,72,32,121,0,32
170  DATA 156,166,32,215,170,104,133,123,104,133,122,32
180  DATA 248,168,76,121,0
190  IF S<>9613
       THEN PRINT "ERROR IN DATA!!":
         END
200  PRINT "OK"
```

If you run the following BASIC program before and after the SYS 828, you can see the difference.

```
100 PRINT "LIST-TEST"
110 LIST 120
120 GOTO 100
```

4.5 Calculated line numbers with GOTO, GOSUB, and RESTORE

Whenever you want to make a program branch or call a subroutine, you must know the exact line number of the point you wish to call. In some cases, it would make programming easier if the line number could be calculated while the program is running, such as the following program assumes.

```
100 PRINT "LINE NUMBER ";L
110 GOTO L
...
```

This could be done with an extensive set of ON ... GOTO statements, and the same applies to the GOSUB command, but it would be much easier if calculated line numbers were allowed.

Another useful extension would be to allow a line number in the RESTORE command. If one has several different blocks of data which are to be read several times, one can only reset the READ/DATA pointer to the beginning of the data and then read over a quantity of unwanted data until the desired data are reached. RESTORE with a line number allows the data pointer to be set to any desired line.

The modification of the GOTO command can be accomplished with a few POKEs. We need only replace the call to get the line number with a routine that will get and evaluate a numeric expression. In doing so we also change the GOSUB routine, since GOSUB calls the GOTO routine.

```
A8A0   20 C0 02    JSR $02C0

02C0   20 8A AD    JSR $AD8A
02C3   4C F7 B7    JMP $B7F7
```

Here we have placed the additional code at $02C0 (704), which is free so long as sprite 11 is not used.

The following BASIC program will place the code in memory:

```
100 FOR I=40960 TO 49151
110 POKE I, PEEK(I) : NEXT
120 A=10*4096+8*256+10*16
130 FOR I=A TO A+2
140 READ X : POKE I,X : NEXT
150 A=704
160 FOR I=A TO A+5
170 READ X : POKE I,X : NEXT
200 DATA 32,192,2
210 DATA 32,138,173,76,247,183
```

Now we have taken care of the GOTO and GOSUB commands. The RESTORE command is somewhat more complicated because there is normally no line number associated with it. We must distinguish between a plain RESTORE command and a RESTORE command with a line number. As it turns out, this is not difficult.

```
02C6 D0 03       BNE $02C8    ;additional characters?
02C8 4C 1D A8    JMP $A81D    ;to old RESTORE command
02CB 20 C0 02    JSR $02C0    ;get line number
02CE 20 13 A6    JSR $A613    ;calculate address of the
                               line number
```

```
02D1 38         SEC
02D2 A5 5F      LDA $5F     ;address low
02D4 E9 01      SBC #$01    ;subtract one
02D6 A4 60      LDY $60     ;address high
02D8 4C 24 A8   JMP $A824   ;continue as per old RESTORE
```

Again, we can place the code in memory with a small BASIC program:

```
300 A=2*256+12*16+6
310 FOR I=A TO A+20
320 READ X : POKE I,X : NEXT
330 DATA 208,3,76,29,168,32,192,2,32,19,166
340 DATA 56,165,95,233,1,164,96,76,36,168
```

Now we must tell the interpreter where the new RESTORE routine is located. Add these lines to the others above:

```
400 POKE 40996, 197 : POKE 40997, 2
410 POKE 1,54
```

Line 410 switches over to the RAM. You can now use the RESTORE command in three different ways:

First, without the line number, as before, second with a line number, or third, with an expression which will yield a line number once evaluated. If a line number is specified, the next READ command will read the first DATA element on the line specified. If this line does not exist, no error message will be given and the pointer will be set to the next line. The following program structure is now possible.

```
100 GOTO 200
...
200 RESTORE 10
...
500 RESTORE
...
800 GOSUB A*2+100
...
900 RESTORE X*100+500
```

4.6 The MID$ command

You are no doubt familiar with the MID$ function, a string function which isolates a part of an alphanumeric string. The following program fragment

```
100 A$ = "TESTSTRING"
110 B$ = MID$(A$,5,3)
120 PRINT B$
```

produces the output

```
STR
```

In this section we offer an enhancement of the MID$ function which allows not only extracting parts of a string but also assignments to parts of a string. With the new command the following type of programming is possible:

```
100 A$ = "TESTSTRING"
110 MID$ (A$,5,3) = "123"
120 PRINT A$
```

Here three characters at position five are replaced with the string "123"; the result is

```
TEST123ING
```

Without this new MID$ command, the string would have to be divided into two parts and then the parts recombined with the string to be inserted:

```
100 A$ = "TESTSTRING"
110 A$ = LEFT$(A$,4) + "123" + MID$(A$,7,3)
120 PRINT A$
```

This command is very useful for replacing individual parts (fields) of a data record. To do this, one defines a string with length equal to the length of the data record and inserts the values of the individual fields with the new MID$ command.

The command is again implemented through a small machine language program.

```
033C                                        ;MID$ AS A PS
EUDOVARIABLE
033C                                        ;
033C                                        ;MID$(STRINGV
ARIABLE,POSITION,LENGTH) = STRINGEXPRESSION
033C                                        ;MID$(STRINGV
ARIABLE,POSITION) = STRING EXPRESSION
033C                                        ;
033C           MIDCOD EQU $CA
033C           EXECUT EQU $308              ;VECTOR FOR S
TATEMENT EXECUTION
033C           CHRGET EQU $73
033C           CHRGOT EQU CHRGET+6
033C           EXECOL EQU $A7E7
033C           VARNAM EQU $45
033C           VARADR EQU $49
033C           DESCRP EQU $64
033C           TESTST EQU $AD8F
033C           GETVAR EQU $B08B
033C           SETSTR EQU $AA52
033C           CHKOPN EQU $AEFA             ;OPEN PARENTH
ESIS
033C           CHKCLO EQU $AEF7             ;CLOSE PARENT
HESIS
033C           CHKCOM EQU $AEFD             ;COMMA
033C           TEST   EQU $AEFF
033C           GETBYT EQU $B79E
033C           FRMEVL EQU $AD9E
033C           ILLQUA EQU $B248
033C           FRESTR EQU $B6A3
033C           LENGTH EQU $03
033C           POSITN EQU $04
033C           VARSTR EQU $05
033C           EQUAL  EQU $B2
```

```
0029 033C          POINT2 EQU $50
0030 033C                                    ;
0031 033C                 ORG 828
0032 033C A9 47    INIT   LDA #<MIDTES
0033 033E A0 03           LDY #>MIDTES
0034 0340 8D 08 03        STA EXECUT
0035 0343 8C 09 03        STY EXECUT+1
0036 0346 60              RTS
0037 0347 20 73 00 MIDTES JSR CHRGET
0038 034A C9 CA           CMP #MIDCOD        ;CODE FOR MID
$
0039 034C F0 06           BEQ MIDSTR         ;YES
0040 034E 20 79 00        JSR CHRGOT
0041 0351 4C E7 A7        JMP EXECOL         ;EXECUTE NORM
AL STATEMENT
0042 0354 20 73 00 MIDSTR JSR CHRGET         ;NEXT CHARACT
ER
0043 0357 20 FA AE        JSR CHKOPN         ;OPEN PAREN
0044 035A 20 8B B0        JSR GETVAR         ;GET VARIABLE
0045 035D 85 64           STA DESCRP
0046 035F 84 65           STY DESCRP+1
0047 0361 85 49           STA VARADR
0048 0363 84 4A           STY VARADR+1
0049 0365 20 A3 B6        JSR FRESTR
0050 0368 A0 00           LDY #0
0051 036A B1 64           LDA (DESCRP),Y
0052 036C 48              PHA                ;LENGTH
0053 036D F0 2E           BEQ ILL
0054 036F 20 52 AA        JSR SETSTR         ;TRANSFER STR
ING TO RAM
0055 0372 A0 01           LDY #1
0056 0374 B1 49           LDA (VARADR),Y
0057 0376 85 05           STA VARSTR         ;SAVE VARIABL
E ADDRESS
0058 0378 C8              INY
0059 0379 B1 49           LDA (VARADR),Y
0060 037B 85 06           STA VARSTR+1
0061 037D 20 FD AE        JSR CHKCOM
0062 0380 20 9E B7        JSR GETBYT         ;GET POSITION
0063 0383 8A              TXA
0064 0384 F0 17           BEQ ILL
0065 0386 CA              DEX
0066 0387 86 04           STX POSITN
0067 0389 20 79 00        JSR CHRGOT
0068 038C C9 29           CMP #')'           ;END OF EXPRE
SSION?
0069 038E D0 04           BNE NEXT
0070 0390 A9 FF           LDA #$FF           ;MAX. LENGTH
0071 0392 D0 0C           BNE STORE
0072 0394 20 FD AE NEXT   JSR CHKCOM
0073 0397 20 9E B7        JSR GETBYT         ;GET LENGTH
0074 039A 8A              TXA
0075 039B D0 03           BNE STORE
0076 039D 4C 48 B2 ILL    JMP ILLQUA
0077 03A0 85 03    STORE  STA LENGTH
```

```
0078 03A2 68                PLA
0079 03A3 38                SEC
0080 03A4 E5 04             SBC POSITN
0081 03A6 C5 03             CMP LENGTH
0082 03A8 B0 02             BCS OK
0083 03AA 85 03             STA LENGTH
0084 03AC 20 F7 AE    OK    JSR CHKCLO         ;CLOSE PAREN
0085 03AF A9 B2             LDA #EQUAL
0086 03B1 20 FF AE          JSR TEST
0087 03B4 20 9E AD          JSR FRMEVL         ;GET EXPRESSI
ON
0088 03B7 20 A3 B6          JSR FRESTR
0089 03BA A0 02             LDY #2
0090 03BC B1 64             LDA (DESCRP),Y
0091 03BE 85 51             STA POINT2+1
0092 03C0 88                DEY
0093 03C1 B1 64             LDA (DESCRP),Y
0094 03C3 85 50             STA POINT2
0095 03C5 88                DEY
0096 03C6 B1 64             LDA (DESCRP),Y
0097 03C8 F0 D3             BEQ ILL            ;ZERO, THEN E
RROR
0098 03CA C5 03             CMP LENGTH
0099 03CC B0 02             BCS OK1
0100 03CE 85 03             STA LENGTH
0101 03D0 A5 05       OK1   LDA VARSTR
0102 03D2 18                CLC
0103 03D3 65 04             ADC POSITN
0104 03D5 85 05             STA VARSTR
0105 03D7 90 04             BCC LOOP
0106 03D9 E6 06             INC VARSTR+1
0107 03DB A4 03             LDY LENGTH
0108 03DD 88          LOOP  DEY
0109 03DE B1 50             LDA (POINT2),Y     ;CHARACTER FR
OM STRING EXPRESSION
0110 03E0 91 05             STA (VARSTR),Y     ;TRANSFER TO
STRING VARIABLE
0111 03E2 C0 00             CPY #0
0112 03E4 D0 F7             BNE LOOP
0113 03E6 4C AE A7          JMP $A7AE          ;TO INTERPRET
ER LOOP

ASSEMBLY COMPLETE.
```

After entering the program, initialize the command expansion by typing

    SYS 828

The following BASIC loader program does the initialization automatically.

```
100  FOR I=828 TO 1000
110    READ X:
        POKE I,X:
        S=S+X:
      NEXT
120  DATA 169,71,160,3,141,8,3,140,9,3,96,32
130  DATA 115,0,201,202,240,6,32,121,0,76,231,167
140  DATA 32,115,0,32,250,174,32,139,176,133,100,132
150  DATA 101,133,73,132,74,32,163,182,160,0,177,100
160  DATA 72,240,46,32,82,170,160,1,177,73,133,5
170  DATA 200,177,73,133,6,32,253,174,32,158,183,138
180  DATA 240,23,202,134,4,32,121,0,201,41,208,4
190  DATA 169,255,208,12,32,253,174,32,158,183,138,208
200  DATA 3,76,72,178,133,3,104,56,229,4,197,3
210  DATA 176,2,133,3,32,247,174,169,178,32,255,174
220  DATA 32,158,173,32,163,182,160,2,177,100,133,81
230  DATA 136,177,100,133,80,136,177,100,240,211,197,3
240  DATA 176,2,133,3,165,5,24,101,4,133,5,144
250  DATA 2,230,6,164,3,136,177,80,145,5,192,0
260  DATA 208,247,76,174,167
270  IF S<>19273
      THEN PRINT "ERROR IN DATA!!":
        END
280  SYS 828:
      PRINT "OK"
```

As an example and test of the new function, try this program:

```
100 DIM A$(10)
110 FOR I = 1 TO 10
120 A$(I) = "TESTSTRING"
130 NEXT
140 FOR I = 1 TO 10
150 MID$ (A$(I),I,1) = MID$("1234567890",I,1)
160 NEXT
170 FOR I = 1 TO 10
180 PRINT A$(I)
190 NEXT
```

The output of the program is ten strings. In the first string, the first character is replaced with a "1", in the second string the second character is replaced with a "2", and so on:

```
1ESTSTRING
T2STSTRING
TE3TSTRING
TES4STRING
TEST5TRING
TESTS6RING
TESTST7ING
TESTSTR8NG
TESTSTRI9G
TESTSTRIN0
```

4.7 INSTR and STRING$ functions

Many other computers have two very useful string functions which the Commodore 64 lacks. The first function, usually called STRING$, creates a string of desired length filled with any given character. The second, often called INSTR, checks to see if a given string is contained within another.

With a knowledge of the BASIC interpreter and the string management of the Commodore 64, it is possible to implement these functions on the 64 as well. We will use the existing command words "POS" and "STR$" for these functions, differentiated from the current BASIC commands with a preceding "!".

The INSTR function has the following syntax:

```
I=!POS(A$,B$,P)
```

A$ is the string to be searched, B$ is the string whose occurrence in A$ you wish to check for, and P is the position at which the search will start. The result is assigned to the variable I, and if zero, then the sought-after string was not found. If the second string was found in the first, I contains the position at which it was found. The input of the position P is optional; if it is not given, the search starts at the beginning of the string. Expressions or functions may be used in place of the variables.

Here are some examples of its use:

```
PRINT !POS("ABCDEFGHIJK","D")
 4

IF !POS(A$,"J") THEN PRINT "FOUND"

A$ = "TESTSTRING"
PRINT !POS(A$,"T")
 1

X = !POS(A$,"T",5) : PRINT X
 6
```

The STRING$ function is used as follows:

```
    A$ = !STR$(L,B)
or
    A$ = !STR$(L,B$)
```

Here A$ is the string which we want to create. L is the length the string will have and B is the ASCII code of the character with which the string will be filled. If a string is used instead of B, the ASCII code of the first character of this string is used. The following examples demonstrate the use of the STRING$ function:

```
PRINT !STR$(20,65)
 AAAAAAAAAAAAAAAAAAAA

A$ = !STR$(10,"*") : PRINT A$
 **********
```

The machine language program is placed in free memory and begins at address 51200.

```
C800                     ORG $C800
C800            CHKOPN EQU $AEFA
C800            CHKCLO EQU $AEF7
C800            CHKCOM EQU $AEFD
C800            FRMEVL EQU $AD9E
C800            CHKSTR EQU $AD8F
C800            FRESTR EQU $B6A3
C800            YFAC   EQU $B3A2
C800            CHRGET EQU $73
C800            CHRGOT EQU CHRGET+6
C800            GETBYT EQU $B79B
C800            INTEGE EQU $B1AA
C800            DESCRP EQU $64
C800            STRADR EQU $62
C800            ADDR2  EQU $FB
C800            ADDR1  EQU $FB+2
C800            LEN1   EQU 3
C800            LEN2   EQU 4
C800            NUMBER EQU 5
C800            START  EQU 6
C800            TYPFLG EQU 13
C800            STRCOD EQU $C4
C800            ILLQUA EQU $B248
C800            SYNTAX EQU $AF08
C800            POSCOD EQU $B9
C800            VECTOR EQU $30A
C800            TEMP   EQU LEN1
C800 A9 0B             LDA #<TESTIN
C802 A0 C8             LDY #>TESTIN
C804 8D 0A 03          STA VECTOR
C807 8C 0B 03          STY VECTOR+1
C80A 60                RTS
C80B A9 00      TESTIN LDA #0
C80D 85 0D             STA TYPFLG
C80F 20 73 00          JSR CHRGET
C812 C9 21             CMP #'!'
C814 F0 06             BEQ TEST2
```

```
0040 C816 20 79 00          JSR CHRGOT
0041 C819 4C 8D AE          JMP $AE8D
0042 C81C 20 73 00   TEST2  JSR CHRGET
0043 C81F C9 B9             CMP #POSCOD
0044 C821 F0 0A             BEQ INSTR
0045 C823 C9 C4             CMP #STRCOD
0046 C825 D0 03             BNE LB1
0047 C827 4C B1 C8          JMP STRING
0048 C82A 4C 08 AF   LB1    JMP SYNTAX
0049 C82D 20 73 00   INSTR  JSR CHRGET
0050 C830 20 FA AE          JSR CHKOPN             ;OPEN PAREN
0051 C833 20 9E AD          JSR FRMEVL             ;GET EXPRESSI
ON
0052 C836 20 8F AD          JSR CHKSTR             ;TEST STRING
0053 C839 A5 64             LDA DESCRP
0054 C83B 48                PHA                    ;STRING ADDRE
SS ON STACK
0055 C83C A5 65             LDA DESCRP+1
0056 C83E 48                PHA
0057 C83F 20 FD AE          JSR CHKCOM             ;COMMA
0058 C842 20 9E AD          JSR FRMEVL             ;SECOND STRIN
G
0059 C845 20 A3 B6          JSR FRESTR
0060 C848 F0 64             BEQ ILL                ;LENGTH=0
0061 C84A 85 04             STA LEN2
0062 C84C 86 FB             STX ADDR2
0063 C84E 84 FC             STY ADDR2+1
0064 C850 68                PLA
0065 C851 A8                TAY
0066 C852 68                PLA                    ;FIRST STRING
 ADDRESS
0067 C853 20 AA B6          JSR FRESTR+7
0068 C856 F0 56             BEQ ILL
0069 C858 85 03             STA LEN1
0070 C85A 86 FD             STX ADDR1
0071 C85C 84 FE             STY ADDR1+1
0072 C85E A2 00             LDX #0                 ;DEFAULT FOR
THIRD PARAMETER
0073 C860 20 79 00          JSR CHRGOT
0074 C863 C9 2C             CMP #','
0075 C865 D0 07             BNE L1
0076 C867 20 9B B7          JSR GETBYT
0077 C86A 8A                TXA                    ;START POSITI
ON
0078 C86B F0 41             BEQ ILL
0079 C86D CA                DEX
0080 C86E 86 06      L1     STX START              ;START POSITI
ON IN STRING
0081 C870 20 F7 AE          JSR CHKCLO
0082 C873 A5 03             LDA LEN1
0083 C875 38                SEC                    ;LEN2>LEN1
0084 C876 E5 04             SBC LEN2
0085 C878 90 28             BCC END                ;RESULT 0?
0086 C87A 69 00             ADC #0
```

```
0087 C87C 85 05              STA NUMBER          ;OF THE SHIFT
S
0088 C87E A5 06              LDA START
0089 C880 18                 CLC                 ;ADDRESS 1 PL
US START POSITION
0090 C881 65 FD              ADC ADDR1
0091 C883 85 FD              STA ADDR1
0092 C885 90 04              BCC L3
0093 C887 E6 FE              INC ADDR1+1
0094 C889 A0 00       L2     LDY #0
0095 C88B B1 FB       L3     LDA (ADDR2),Y
0096 C88D D1 FD              CMP (ADDR1),Y       ;COMPARE CHAR
ACTERS
0097 C88F D0 0B              BNE L5              ;SEARCH AT NE
XT POSITION
0098 C891 C8                 INY
0099 C892 C4 04              CPY LEN2            ;ALL CHARACTE
RS OF STRING2 TESTED?
0100 C894 90 F5              BCC L3
0101 C896 A4 06              LDY START
0102 C898 C8                 INY
0103 C899 4C A2 B3    L4     JMP YFAC            ;RESULT
0104 C89C E6 06       L5     INC START
0105 C89E C6 05              DEC NUMBER
0106 C8A0 D0 04              BNE L6              ;NOT DONE?
0107 C8A2 A0 00       END    LDY #0              ;NOT FOUND, Z
ERO
0108 C8A4 F0 F3              BEQ L4
0109 C8A6 E6 FD       L6     INC ADDR1
0110 C8A8 D0 DF              BNE L2              ;INCREMENT ST
RING2 ADDRESS
0111 C8AA E6 FE              INC ADDR1+1
0112 C8AC D0 DB              BNE L2
0113 C8AE 4C 48 B2    ILL    JMP ILLQUA
0114 C8B1                                        ;
0115 C8B1                                        ;STRING$ FUNC
TION
0116 C8B1                                        ;
0117 C8B1 20 73 00    STRING JSR CHRGET
0118 C8B4 20 FA AE           JSR CHKOPN          ;OPEN PAREN
0119 C8B7 20 9E B7           JSR GETBYT+3
0120 C8BA 8A                 TXA
0121 C8BB 48                 PHA                 ;SAVE LENGTH
0122 C8BC 20 FD AE           JSR CHKCOM
0123 C8BF 20 9E AD           JSR FRMEVL
0124 C8C2 24 0D              BIT TYPFLG
0125 C8C4 30 0C              BMI STR             ;STRING
0126 C8C6 20 AA B1           JSR INTEGE
0127 C8C9 A5 64              LDA DESCRP          ;HIGH BYTE
0128 C8CB D0 E1              BNE ILL             ;>255
0129 C8CD A5 65              LDA DESCRP+1        ;LOW BYTE, LE
NGTH
0130 C8CF 4C DB C8           JMP STR2
0131 C8D2 20 82 B7    STR    JSR $B782           ;SETSTR, TYPF
LG TO NUMERIC
```

```
0132 C8D5 F0 D7             BEQ ILL             ;LENGTH ZERO
0133 C8D7 A0 00             LDY #0
0134 C8D9 B1 22             LDA ($22),Y         ;FIRST CHARAC
TER
0135 C8DB 85 03       STR2  STA TEMP
0136 C8DD 68                PLA                 ;LENGTH
0137 C8DE 20 7D B4          JSR $B47D           ;FRESTR
0138 C8E1 A8                TAY
0139 C8E2 F0 07             BEQ STR3
0140 C8E4 A5 03             LDA TEMP
0141 C8E6 88          LOOP  DEY
0142 C8E7 91 62             STA (STRADR),Y      ;CREATE STRIN
G
0143 C8E9 D0 FB             BNE LOOP
0144 C8EB 20 CA B4    STR3  JSR $B4CA           ;PUT STRING I
N DESCRIPTOR STACK
0145 C8EE 4C F7 AE          JMP CHKCLO

ASSEMBLY COMPLETE.
```

```
100   FOR I=51200 TO 51440
110     READ X:
          POKE I,X:
          S=S+X:
        NEXT
120   DATA 169,11,160,200,141,10,3,140,11,3,96,169
130   DATA 0,133,13,32,115,0,201,33,240,6,32,121
140   DATA 0,76,141,174,32,115,0,201,185,240,10,201
150   DATA 196,208,3,76,177,200,76,8,175,32,115,0
160   DATA 32,250,174,32,158,173,32,143,173,165,100,72
170   DATA 165,101,72,32,253,174,32,158,173,32,163,182
180   DATA 240,100,133,4,134,251,132,252,104,168,104,32
190   DATA 170,182,240,86,133,3,134,253,132,254,162,0
200   DATA 32,121,0,201,44,208,7,32,155,183,138,240
210   DATA 65,202,134,6,32,247,174,165,3,56,229,4
220   DATA 144,40,105,0,133,5,165,6,24,101,253,133
230   DATA 253,144,2,230,254,160,0,177,251,209,253,208
240   DATA 11,200,196,4,144,245,164,6,200,76,162,179
250   DATA 230,6,198,5,208,4,160,0,240,243,230,253
260   DATA 208,223,230,254,208,219,76,72,178,32,115,0
270   DATA 32,250,174,32,158,183,138,72,32,253,174,32
280   DATA 158,173,36,13,48,12,32,170,177,165,100,208
290   DATA 225,165,101,76,219,200,32,130,183,240,215,160
300   DATA 0,177,34,133,3,104,32,125,180,168,240,7
310   DATA 165,3,136,145,98,208,251,32,202,180,76,247
320   DATA 174
330   IF S<>30119
        THEN PRINT "ERROR IN DATA!!":
          END
340   SYS 51200:
        PRINT "OK"
```

4.8 Automatic line numbering

In this section we want to present a useful command for the Commodore 64 which makes it much easier to enter programs. This command, similar to the "AUTO" command found on other computers, will automatically create line numbers for you so that you do not have to type them in yourself. You can set both the starting line number and the increment by which each successive line number will be increased. It is quite simple to use this new command:

To turn on the automatic line numbering, enter the following command:

```
SYS 828, startnumber, increment
```

Ex. SYS 828, 100, 10

The increment may be an integer value up to 255. After entering the SYS command, the first line number is printed and the cursor is placed behind it. You can enter the program line directly and press RETURN when done. Now the next line number will be displayed automatically, line 110 in our example.

```
100 INPUT "INPUT";A$
110
```

To end the AUTO command, simply press RETURN without typing anything else on a line. If you later want to continue entering lines, you need only enter

SYS 828

The line number at which you left off will automatically be displayed. You can of course change the starting line number and increment at any time by entering these along with the SYS command.

The machine language program is stored in the cassette buffer. Following this assembly language listing is again a loader program in BASIC.

```
0001 033C                    ORG 828            ;CASSETTE BUF
FER
0002 033C          LO        EQU $14
0003 033C          HI        EQU LO+1
0004 033C          FAC       EQU $62            ;FLOATING-POI
NT ACCUMULATOR
0005 033C          CR        EQU 13             ;CARRIAGE RET
URN
0006 033C          LINE      EQU 251            ;LINE
0007 033C          INCR      EQU LINE+2         ;INCREMENT
0008 033C          INTFLT    EQU $BC49          ;INTEGER TO F
LOATING POINT
0009 033C          FLTASC    EQU $BDDD          ;FLOATING POI
NT O ASCII
0010 033C          VECTOR    EQU $302           ;LINE INPUT
0011 033C          INPUT     EQU $FFCF
0012 033C          PRINT     EQU $FFD2
0013 033C          BUFF1     EQU $101
0014 033C          BUFF2     EQU $200
0015 033C          MNLOOP    EQU $A486
0016 033C          GOON      EQU $A569
0017 033C          CONT      EQU $A576
0018 033C          CHRGOT    EQU $79
0019 033C          OLDVEC    EQU $A483
0020 033C          GETPAR    EQU $B7EB
0021 033C          CHKCOM    EQU $AEFD
0022 033C 20 79 00           JSR CHRGOT         ;ADDITIONAL C
HARACTERS?
0023 033F F0 10              BEQ LO             ;NO
0024 0341 20 FD AE           JSR CHKCOM         ;COMMA?
0025 0344 20 EB B7           JSR GETPAR         ;GET PARAMETE
R
0026 0347 86 FD              STX INCR
0027 0349 A5 14              LDA LO             ;AND SAVE
0028 034B 85 FB              STA LINE
0029 034D A5 15              LDA HI
```

```
034F 85 FC            STA LINE+1
0351 A9 5C      LO    LDA #<AUTO
0353 8D 02 03         STA VECTOR        ;SET INPUT VE
CTOR
0356 A9 03            LDA #>AUTO
0358 8D 03 03         STA VECTOR+1
035B 60               RTS
035C                                    ;
035C 20 62 03   AUTO  JSR AUTNUM
035F 4C 86 A4         JMP MNLOOP
0362                                    ;
0362 A5 FB      AUTNUM LDA LINE
0364 A6 FC            LDX LINE+1
0366 85 63            STA FAC+1         ;LINE NUMBER
0368 86 62            STX FAC
036A A2 90            LDX #$90
036C 38               SEC
036D 20 49 BC         JSR INTFLT        ;TO FLOATING
POINT
0370 20 DD BD         JSR FLTASC        ;TO ASCII
0373 A2 00            LDX #0
0375 BD 01 01   L1    LDA BUFF1,X       ;GET DIGITS
0378 F0 09            BEQ L2
037A 9D 00 02         STA BUFF2,X       ;IN BASIC BUF
FER
037D 20 D2 FF         JSR PRINT         ;AND OUTPUT
0380 E8               INX
0381 D0 F2            BNE L1
0383 A5 FB      L2    LDA LINE          ;LINE NUMBER
0385 18               CLC
0386 65 FD            ADC INCR
0388 85 FB            STA LINE
038A 90 02            BCC L3
038C E6 FC            INC LINE+1
038E A9 20      L3    LDA #32           ;OUTPUT SPACE
0390 20 D2 FF         JSR PRINT
0393 20 CF FF         JSR INPUT
0396 C9 0D            CMP #CR           ;EMPTY INPUT
0398 F0 03            BEQ L4            ;YES
039A 4C 69 A5         JMP GOON          ;CONTINUE
039D A5 FB      L4    LDA LINE
039F E5 FD            SBC INCR          ;NEXT LINE NU
MBER
03A1 85 FB            STA LINE
03A3 B0 02            BCS L5
03A5 C6 FC            DEC LINE+1
03A7 A9 83      L5    LDA #<OLDVEC
03A9 A0 A4            LDY #>OLDVEC
03AB 8D 02 03         STA VECTOR        ;PUT OLD
03AE 8C 03 03         STY VECTOR+1      ;VECTOR BACK
03B1 4C 76 A5         JMP CONT

ASSEMBLY COMPLETE.
```

```
100 FOR I = 828 TO 947
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA  32,121,  0,240, 16, 32,253,174, 32,235,183,134
130 DATA 253,165, 20,133,251,165, 21,133,252,169, 92,141
140 DATA   2,  3,169,  3,141,  3,  3, 96, 32, 98,  3, 76
150 DATA 134,164,165,251,166,252,133, 99,134, 98,162,144
160 DATA  56, 32, 73,188, 32,221,189,162,  0,189,  1,  1
170 DATA 240,  9,157,  0,  2, 32,210,255,232,208,242,165
180 DATA 251, 24,101,253,133,251,144,  2,230,252,169, 32
190 DATA  32,210,255, 32,207,255,201, 13,240,  3, 76,105
200 DATA 165,165,251,229,253,133,251,176,  2,198,252,169
210 DATA 131,160,164,141,  2,  3,140,  3,  3, 76,118,165
220 IF S <> 15495 THEN PRINT "ERROR IN DATA!!" : END
230 PRINT "OK"
```

4.9 User-defined functions--DEF FN

Many programmers prefer to add more lines or subroutines to their program instead of simply defining their functions. Admittedly, this technique is not well-described in the user's guide, so we will try to clarify its use. In addition to the lack of emphasis in the documentation, there is another reason why this technique is not used: The function does not appear very powerful, at least at first glance, because only one argument may be passed to the user-defined functions.

In this section we want first to illustrate the use of these functions and second, to show how complex multi-variable formulas can be implemented by nesting several functions.

A function definition is constructed in the following manner:

```
DEF FN name (function variable) = arithmetic expression
```

Example:

```
DEF FN A(X) = 2*X + B
```

Our function has the name A and function variable X. When the function is called, the expression between the parentheses (it need not be X or even a variable) will be substituted for X in the arithmetic expression. Any variables with the name A or X will remain undisturbed. The function variable X is used only as a place holder for the actual value given in the function call. In contrast, B implies the actual variable B which must be defined before calling the function. The function variable is often described as the dummy variable. The result of the function must always be numeric--a string expression is not allowed.

As an example, we will define a formula which will round-off a value to the nearest hundreth--a function useful for working with dollar amounts. As you know, the third decimal place determines the rounding. If this value is 5 or greater, the second decimal place is increased by one (rounded up), else the number is merely truncated at the second decimal place (rounded down).

Most programmers would place this rounding function directly into the program:

```
A = INT (B*100+.5)/100
```

If this function must be used more than once, one can save time and space by replacing it with a function call

```
A = FNX(B)
```

First, however, the function must be defined:

```
 10 DEF FN R(B) = INT(B*100+.5)/100
    ...
100 A = FNR(B)
```

The command in line 100 can be used as often as necessary in place of the longer formula, in which the variable B contains the value to be rounded.

Now an example of nesting functions. Here we will calculate the price of an item based on the cost, a given profit margin, and the sales tax.

C is the cost
P is the profit margin in %
S is the sales tax in %

```
10 DEF FN SL(C) = (C/(1-P/100)*(1+S/100))
20 DEF FN PR(B) = INT (FN SL(B)*100+.5)/100
   ...
100 A = FN PR(C)
```

After line 100, A contains the retail price, rounded to the nearest penny. Functions may also be nested deeper, of course. A maximum of ten functions may be nested. If you try to nest them any deeper, an "OUT OF MEMORY ERROR" will occur, indicating that the stack has overflowed (not necessarily that the BASIC program storage was exceeded). If subroutines (GOSUB) or FOR-NEXT loops are active during a FN call, the total nesting level including subroutines, FOR-NEXT loops, and FN levels must not exceed the maximum of ten levels.

4.10 BASIC HARDCOPY routine

Have you ever tried to print the contents of the screen on your printer? There are a number of machine language program that allow you to do this, but it is also easy to do in BASIC.

Since you must understand the operation of the program, you must first learn something about the construction of the screen memory.

As you know, there are 1000 characters at your disposal on the screen. These 1000 characters are organized as 25 lines of 40 characters each. These characters are naturally not only on the video display, but also stored in the Commodore 64's memory. Normally, the area in which you will find the individual characters is in RAM from address 1024 to 2023. In order to place on the paper all of the information on the screen, we must read the characters out of this memory area with PEEK and then print the corresponding values with the CHR$ function. It is important to note that the values in the range 0 to 31 cannot be printed directly because these values belong to the ASCII range of control characters and are by nature non-printable.

There is one other thing we must pay attention to. The lines on the screen are 40 characters long but a line on a printer normally consists of at least 80 characters. If you are familiar with programming in BASIC, you know that to print characters one after each other with multiple PRINT statements (screen or printer) requires that the PRINT statements be followed with a semicolon. For our hardcopy routine we must print 40 characters one after the other (one entire screen line) and then send a carriage return to the printer so that the screen image is printed the same way it appears on the screen. We can accomplish this with nested loops. The completed program looks like this:

```
50000 OPEN 4,4: REM OPEN PRINTER CHANNEL FOR UPPER/GRAPHICS
50010 FOR I=1024 TO 1984 STEP 40: REM 25 LINES
50015 BL$="": REM ERASE LINE
50020 FOR J=0 TO 39: REM 40 LINES
50030 L=PEEK(I+J): REM READ CHARACTER
50040 IF L<32 THEN L=L+64: REM CONVERT TO UPPER CASE
50050 BL$=BL$+CHR$(L): REM BUILD LINE
50060 NEXT J : REM NEXT CHARACTER
50070 PRINT BL$: REM PRINT LINE
50080 NEXT I: REM NEXT LINE
50090 RETURN: REM BACK TO MAIN PROGRAM
```

You may have noticed that we wrote this program slightly differently than we had discussed before. If you have seen a hardcopy routine in action, you may have noticed that a certain amount of time goes by before the printer actually prints the line after it has been sent. Why? Almost every printer works with something called a buffer (of varying size). The characters which the computer sends are placed into this buffer until it is full. Then the buffer is printed. The advantage of this is that an entire line can be printed faster than just one character at a time. It is for a similar reason that we first fill a text string (BL$) with the individual characters before we print it. The computer is able to send a 40-character string faster than it can send 40 individual characters.

One other feature of this program is the conversion to upper case characters in line 50040. The upper case characters in the Commodore 64 are stored in memory using a range of codes starting with 1. In standard ASCII however, this character range starts with 65, therefore, all numbers less than 32 are incremented by 64 to allow them to be printed correctly.

Chapter 5 : Forth

5.1 Programming in Forth

What is Forth? This is a question often asked by those who have only programmed in BASIC or assembly language up to this point. Certainly many of you will say "Why should I learn another programming language when my Commodore has a good version of BASIC?" This objection may seem justified at first, but after a closer look, one must consider if it really is justified, especially when a computer can speak more than one language.

Once you have programmed in BASIC for a while, you will come across things which either cannot be done at all or are very difficult to do. On larger computers, one has the ability to switch to other languages. There is FORTRAN for mathematical applications, COBOL for commercial purposes, assembly language for time-critical tasks, BASIC for general problem solving and so on. Then there are languages designed to force structured programming such as Pascal and ELAN. Each language has its strengths and weaknesses.

Forth belongs to the youngest generation of programming languages, as its name says, to the fourth generation. The developers of Forth have tried to implement all the advantages of the older, better-known languages without the disadvantages of these languages.

Forth has in its structure some very striking advantages, especially for microcomputers:

1. The computer on which Forth runs requires a very small address space. Because Forth programs do not require much space, large, efficient programs can be created on a very small computer.
2. Forth is ideally suited for performing low-level (machine-level) or I/O functions even though one need not be acquainted with the hardware of the device in any great detail in order to program in Forth. It is often used in industrial control applications and robotics.
3. In addition to the first two advantages, Forth does not require a disk drive, although it is a good idea to have one.

Forth consists of five parts:

1. DICTIONARY: The philosophy of Forth is such that the set of commands in Forth which relate directly to machine language code is very small. The user is permitted to define his own commands and use these in subsequent programs. This allows you to personalize Forth or optimize for performing certain types of operations. The dictionary itself is a linked list containing the current Forth commands (called words) and information necessary to execute them.

2. STACK: The stack is the most important element of Forth. The notion of a stack is familiar to anyone who has done any machine language

programming or who owns a Reverse Polish Notation (RPN) calculator. We will come back to this later.

The stack uses the last-in/first-out (LIFO) method of data storage. Virtually all operations work with the stack.

3. INTERPRETER: Forth, like BASIC, is an interpreter. This means that one first creates a Forth program with the editor and then starts the program with the appropriate command. Error checking occurs while the program is running. There is no tedious waiting while the program compiles; it can be started immediately. This has the result that interpreted programs are typically slower than compiled programs. The time factor difference is not as great with Forth. The interpreter is divided into a text interpreter and an address interpreter. The text interpreter checks the words in the dictionary, and when the word is found, the address interpreter is activated. This interpreter works with absolute addresses, calling in turn each of the words which make up a user-defined or higher-level Forth word. These addresses are "compiled" into a word's dictionary entry at the time it is defined.

4. ASSEMBLER: Many Forth interpreters contain an assembler. This assembler can be used to define words which will then execute machine language routines when called. This method of programming is sometimes required for I/O--establishing

contact with the external world. Forth itself bears a certain resemblance to assembly language; it is quite fast, but far easier to learn.

5. **MEMORY:** Memory is important for any programming language, and no less so for Forth, although it typically requires far less than other programming languages. In Forth, memory can be treated like blocks on a disk drive, and, to a certain extent, blocks on a disk drive can be treated like memory.

5.2 A comparison of Forth and BASIC

The best way to see the advantages of Forth is to compare two programs, one written in Forth and the other in BASIC, which perform the same task. Before we present these programs, we must clarify a few things.

In section 5.1 we mentioned that the stack plays a very important role in Forth, and that one can compare it to the method of operation of an RPN calculator (such as those made by Hewlett-Packard):

Let's calculate (2 + 3) * (4 + 5) on an HP calculator. The keys we press are:

```
2 <ENTER> 3 + 4 <ENTER> 5 + *
```

This looks confusing at first, but it is necessary in order to solve the equation without using parentheses. Pressing "2" and then the ENTER key places the number 2 on the top of the stack. Pressing "3" places this number on the stack, first moving the 2 one place lower on the stack. The stack now looks like this:

```
STACK:          TOP    3
                       2
                       -
                       -
                       -
                       -
              BOTTOM   -
```

After the "+" key is pressed, the addition is carried out. The "+" operation removes the top two values from the stack, adds them, and them pushes the result back onto the stack. The stack now looks like this:

```
        TOP     5
                -
                -
                -
                -
                -
     BOTTOM     -
```

Pressing the "4" key pushes the 5 down one place and puts the 4 on top:

```
        TOP     4
                5
                -
                -
                -
                -
     BOTTOM     -
```

Entering the number 5 moves the old 5 and the 4 one place down on the stack:

```
        TOP     5
                4
                5
                -
                -
                -
     BOTTOM     -
```

The "+" operation again removes the two most recently entered values from the stack, adds them and pushes the result back onto the stack.

```
        TOP     9
                5
                -
                -
                -
                -
     BOTTOM     -
```

The last operation is the multiplication. It works in the same way as the addition.

```
        TOP     45
                -
                -
                -
                -
                -
     BOTTOM     -
```

Now the result is at our disposal. This process seems quite complicated and time-consuming, but each calculator and computer works on this principle. On Hewlett-Packard calculators, as in Forth, this process is made explicit. Those who have done some programming in assembly language will be able to learn Forth with few difficulties, but even the novice learning Forth will have fewer problems than he might think. The greatest obstacle to learning Forth is that it is so different from most other languages, not that it is so difficult to understand on its own.

We will now present a small Forth program which will clarify the operation of the stack and also illustrate the process of defining new words and adding them to the Forth vocabulary. The program takes the cube of a number; since there is no command to perform this calculation, we must define one:

```
: CUBE          ( THIS WORD IS BEING DEFINED )
     DUP DUP    ( COPY THE NUMBER TWICE ON STACK )
     * *        ( MULTIPLIES TOP TWO STACK VALUES 2X )
     ;
```

The colon in Forth tells the interpreter that a word is being defined. If we had not entered the colon, Forth would have responded

```
CUBE ?
```

indicating that it had not found this word in its dictionary. Since we did use the colon, however, Forth will treat everything up to the semicolon as the definition of this word. After encountering this character, Forth replies

```
OK
```

The word CUBE is now part of our Forth dictionary and we can use it directly or within a program. It will remain so only as long as the computer is turned on, unless we save it onto a disk.

The command DUP makes a copy of the value at the top of the stack and puts this copy back on the stack. Since we want not the square but the cube of the number, we must make two copies of the number. If the number 5 was at the top of the stack, the stack would now look like this:

```
        TOP    5
               5
               5
               -
               -
               -
     BOTTOM    -
```

Now we must multiply the numbers together. A total of two multiplications are necessary. Forth uses the usual "*" symbol for multiplication. After we have performed the multiplications, the cube will be at the top of the stack, and we can end our definition. Note that during a colon definition none of these operations are actually carried out. The colon places Forth in what is called the compile mode, where it searches for each word in the definition and makes note of that word's address within the dictionary, which it places in the definition for the new word. When we now use this new word, a colon run-time routine calls each of the words in turn, thereby executing the new command. Here are some examples of our new command (the "." tells

Forth to print the value at the top of the stack):

You enter:	Forth responds:
5 CUBE .	125 OK
1 CUBE .	1 OK
-15 CUBE .	-3375 OK

As you can see, it is very easy to add new commands to Forth. You can make use of this feature to optimize the language to a specific application or set of applications.

Now let's compare Forth and BASIC. The program we will use calculates the cubes of the integers from zero to ten. The Forth program will make use of our newly defined word CUBE.

1. Forth

```
: CUBENUMBERS
     10 0              ( FROM 0 TO 10 * LIFO!! * )
     DO                ( START OF LOOP )
        CR I . I CUBE .
                       ( PRINT NUMBER (I) AND CUBE )
     LOOP              ( END OF LOOP )
     ;

CUBENUMBERS
0 0
1 1
2 8
3 27
```

```
4 64
5 125
6 216
7 343
8 512
9 729  OK
```

2. BASIC

```
10 REM CALCULATE CUBES
20 MIN=0 : MAX=9
30 FOR I=MIN TO MAX : PRINT I,I*I*I
40 NEXT I
50 END

RUN
0    0
1    1
2    8
3    27
4    64
5    125
6    216
7    343
8    512
9    729
READY.
```

Both programs require approximately the same number of lines, but the Forth program requires far less storage space than does the BASIC program. Efficient use of memory is not everything, however. Let us compare the speeds of Forth and BASIC. We can do this with a simple loop:

1. Forth

```
: BENCHMARK
     30000 0          ( FROM 0 TO 30000 )
     DO               ( START OF LOOP )
                      ( EMPTY LOOP )
     LOOP             ( END OF LOOP )
     ;                ( END OF DEFINITION )

BENCHMARK  OK
```

2. BASIC

```
10 REM  BENCHMARK
20 MIN=0 : MAX=30000
30 FOR I=MIN TO MAX
40 NEXT I
50 END

RUN
READY.
```

The results may be quite surprising:

Language:	Time:
BASIC	about 40 seconds
Forth	about 4 seconds

Remember, these test were performed the same computer, the Commodore 64.

This advantage alone should prompt many people to give serious consideration to learning Forth. Programming in it is quite easy and the speed and memory savings are significant.

Forth - The language for professional software developers

It is interesting to note that more and more professional software developers are changing their minds about Forth because of the many advantages that we have already discussed. In addition, Forth offers a short development time because it is as structured language, is easy to use, and in spite of this still offers the flexibility and speed of machine language. There are already programs for the Commodore 64 which have been developed in Forth, such as the spreadsheet program Calc Result.

Forth has one last advantage which is of special interest to software houses. It belongs to the small group of portable languages, which means that a program written in Forth on one computer can easily be made to run on a different computer. This reduces the time required to produce a given software packages for multiple computers, something which is very important to software companies.

If you are interested in learning more about Forth, you can try our TINY FORTH package, available for the Commodore 64 or VIC-20.

Chapter 6 : CP/M on the Commodore 64

6.1 Introduction to CP/M

CP/M is one of the most widely-used microcomputer operating system. It has become the "standard" operating system, inasmuch as such a thing exists. CP/M has withstood the test of time, something which cannot be said of many other microcomputer operating systems. Most of the bugs have been worked out and the system is reasonably trustworthy.

What can the Commodore 64 user gain from this operating system? He is used to BASIC 2.0 and the Commodore DOS, why another operating system? This question is not often found outside of Commodore users who have not seen much of the rest of the computer world. There one finds an undreamed-of quantity of software. Not that the 64 does not have a significant amount of software available for it, but it is nothing compared to the sheer volume available for CP/M.

Not only can the user profit from the availability of CP/M software, but the programmer as well. He can write his software for a much larger body of users than ever before possible. Writing a program specifically for the Commodore 64 limits the potential users to 64 owners, but many different computers can use CP/M, so writing a program to run under CP/M greatly increases the potential market for a program. Many programmers writing for an "exotic" operating system have heard "And when will the program be available on CP/M?"

At this point we must point out one major problem with every CP/M system:

CP/M IS NOT THE SAME AS CP/M!

Unfortunately, most computer manufacturers use their own modified version of CP/M. Despite the apparent compatibilty, it is not possible to interchange programs or transfer data. The CP/M for the Commodore 64 also has its peculiarities. For example, the I/O byte of CP/M is not implemented, the 64 can display only 40 characters per line on the video display, and even the disk format is not compatible with other computers. We will return to these problems and how to solve them later.

CP/M has certain standards, some of which we have already mentioned:

1. The computer on which it runs must have at least 48K of RAM.

2. CP/M occupies the free memory at address $0100.

3. Most programs require a video display capable of displaying 24 (or 25) lines of 80 characters.

4. Much CP/M software is available only on 8 inch disks.

Let's take a brief look at these standards. First, the computer must have at least 48K of RAM. Virtually all currently produced computers can be expanded to 48K of RAM, and the Commodore 64 already comes with more than this. The ability of the 64 to switch the ROMs out is also important for implementing CP/M and other programs which could not be used on computers without this capability. CP/M can be placed where it is supposed to go, at address $0100.

The first problem with implementing CP/M on the 64 is the limited screen size. CP/M programs such as Wordstar, Datastar, and others require an 80 column display for proper operation. The Commodore 64 has only 40 characters, although there is a solution which we will say more about in section 6.3.

The last problem concerns the disk drive. We mentioned that much of the CP/M software is available on 8 inch disks. The 1541 disk drive uses 5 1/4 inch disks. More and more computer manufacturers are using the 5 1/4 inch disk drives and so more CP/M software is being made available for these formats. Unfortunately, none is yet available in the 1541 format.

What does CP/M consist of?

CP/M is an operating system composed of several parts. More exactly, it consists of four major parts.

1. BIOS (Basic Input/Output System): As the name implies, the BIOS is concerned with communicating with the outside world. It is used to send information to the printer, the terminal (screen), to the disk drive, and so on. The BIOS has a number of function calls which tell the operating system how this communication will take

place. A complete table of these functions is found at the end of this description.

2. BDOS (Basic Disk Operating System): This part controls the disk drives--the management of the directory and the actual read and write commands. These procedures are also controlled by individual function calls.

3. CCP (Console Command Processor): The operating system must be told what it is you want it to do. This is generally done via the Commodore 64's keyboard. The CCP transmits your commands to the CP/M system.

4. TPA (Transient Program Area): This is the free program area which is available to the user. This storage area is used when you write or use a program.

This is the layout of the BIOS, BDOS, CCP, and TPA in memory:

Name:	Address:
FDOS (BIOS + BDOS)	$9C00
CCP	$9400
TPA	$0100
System parameters	$0000

Here is the table of FDOS functions:

Number:	Function:
	BIOS
00	System reset
01	Read ASCII character from terminal
02	Send ASCII character to terminal
03	Read ASCII character from paper tape reader
04	Send ASCII character to paper tape punch
05	Send ASCII character to printer
06	Send/receive character to/from console
07	Read status from device
08	Send status to device
09	Send character string buffer
10	Read character string into buffer
11	Read status of console
	BDOS
12	Read CP/M version number
13	Disk reset
14	Select drive number
15	Open file (OPEN)
16	Close file (CLOSE)
17	Search for first program in FCB
18	Search for next program in FCB
19	Erase program (DELETE)
20	Read from sequential file

21	Write to sequential file
22	Create file
23	Change filename (RENAME)
24	Input possible drives
25	Read current drive number
26	Set DMA address
27	Read address
28	Set write protect
29	Read read/write pointer
30	Set read/write pointer
31	Read address of disk parameters
32	Read/set user id
33	Read from random file
34	Write to random file
35	Calculate program length
36	Read address of record

All of these functions are called in a specific pattern. In order to clarify this, we must learn a little bit about 8080 machine language. Because CP/M was developed on this processor and the Z-80 microprocessor which is found in the Commodore CP/M module also understands the 8080 machine language, CP/M applications are written in it. If you are not familiar with this machine language, but you would like to delve deeper into CP/M, we strongly recommend that you get a CP/M handbook such as Rodney Zaks CP/M Handbook and a good book on Z-80 or 8080 machine language.

What do these function calls look like?

As an example, we will read the version number of the Commodore CP/M (2.2) with the following routine:

```
MVI     C,12     ;FUNCTION 12
CALL    0005     ;JUMP TO BDOS
CPI     20H      ;$20 INDICATES CP/M 2.0
.       .
.       .
```

First the C register is loaded with the value 12, the function number for reading the CP/M version number. This C register is always loaded with the function number before the branch is made to address 0005. CP/M now knows that we want to find out the version number, and branches automatically to the point in the CP/M system where the version number will be read. BDOS then jumps back to our routine after placing the version number in the accumulator. If the value is $20 then we know that the version number is at least 2.0. This information is very important if we want to write a program which uses random file access because all CP/M versions before 2.0 can work only with sequential files (see chapter 8). If we want our program to run on a CP/M computer other than the Commodore 64, we can easily determine if it will run or not by reading the version number and checking to see that it is 2.0 or greater.

The various registers play an important role in the CP/M function calls. The first to mention is the C register which contains the function number prior to the BDOS/BIOS call. After the execution of the appropriate routine, the other registers contain the desired information. Some functions do not return any information, rather they output information to some device, or inform the operating system itself of something. For these types of calls, the appropriate register or registers are loaded with the information, the C register is loaded with the function code, and then call is made.

Why function codes?

As you know by know, it is an advantage of the CP/M operating system that a CP/M program can generally be run on any CP/M computer. Small changes to the BIOS/BDOS or CCP may be necessary to implement CP/M itself on different computers, however. In order to guarantee that a program will run without the programmer having to worry about the construction of a particular version of CP/M, the function calls to the BIOS/BDOS via address 0005 are used. This way a given part of the operating system can be changed without having to rewrite any programs. The same advantage is found in the kernal ROM of the Commodore 64, without the CP/M operating system. In the kernal is a list of subroutine entry points called a vector table which call the various input/output routines. If any of these routines are changed, it is still possible to use the old programs; they notice nothing of the altered operating system.

The CCP commands

The CCP serves as an interface between the user and the CP/M operating system. Programs can be executed from the CCP and it also supervises its own small set of commands:

1. DIR (DIRectory): This command displays the contents of the disk. It offers the following options:

   - DIR displays the entire directory listing of the disk in the currently selected drive.

- DIR B: displays the entire directory listing of the diskette in drive B (1). "A" may be used in place of B, causing the directory of the disk in drive A (0) to be displayed.
- DIR <name.ext> only indicates whether or not the given file is on the diskette. The name may be up to eight characters long, must start with a letter, and may contain no special characters (punctuation) except the extension separator, the period. "ext" is a three-letter extension of the program name. It is normally used to indicate the type of program. For example, only programs which end in COM may be executed directly. TXT indicates that the file contains text, BAS indicates a BASIC program, and so on.
- DIR <*.ext> displays all programs ending with ext. DIR *.COM would display all programs ending with .COM, i.e. all directly-executable programs.

See the CP/M manual for other options with the DIR command.

Here is the format of the directory listing when using the DIR command on the Commodore 64:

```
A>DIR
A: MOVCPM   COM : PIP      COM
A: SUBMIT   COM : XSUB     COM
A: ED       COM : ASM      COM
A: DDT      COM : LOAD     COM
```

```
A: STAT      COM : SYSGEN   COM
A: DUMP      COM : DUMP     ASM
A: COPY      COP : CONFIG   COM
A>
```

2. ERA (ERAse): This command erases one program or several programs from the directory. Here too there are several options:

-ERA <name.ext>

-ERA <*.ext>

3. REN (REName): With this command you can give an existing program or data file a new name. There is only one form of the command:

- REN <new name> = <old name>

4. TYPE: This command is used only for text files. It displays the contents of a file on the screen. It has the form:

- TYPE <filename.ext>

In most cases the extension will be TXT, PRN, or something similar.

5. SAVE: This command appears quite complicated at first. It is normally used to save a program modified with DDT, or for saving a modified CP/M version. The format is:

- SAVE <number of pages> <name.ext>

The number of pages is the number of 256-byte "pages" to be saved. The command

SAVE 50 TEST.COM

puts the contents of memory from address $0100 (start of the TPA) to address $32FF under the name TEST.COM. The length is 50*256.

6. USER: This command allows the directory to be divided up for different users. It is possible to protect certain areas of the directory from access by other users (on other computers). This command has no real value on the Commodore 64, however. The form is:

- USER user number

The user number is an integer from 0 to 15. Entering the number places one in the directory of the corresponding user. The default number is zero.

These are the commands which every computer using the CP/M operating system understands. Only the USER command is new since version 2.0. All others belong to the standard CP/M command set. In the next pages we will present a short overview of the standard CP/M programs: PIP, ED, DDT, and STAT. They are supplied on every CP/M system diskette, but they are programs, not commands. They serve to expand the commands available on CP/M, but they must be first loaded in from disk.

6.2 The individual CP/M programs

What would CP/M be without its framework of utility programs? Digital Research, the producer of CP/M, has made sure that the user can start programming immediately. (Note to Commodore 64 users: No version of BASIC comes with the Commodore CP/M although one may be added later).

The following programs belong to the CP/M standard:

- STAT.COM — A program which obtains and displays the various system information such as the space left on the disk, device assignments, and so on.

- ASM.COM — This is an assembler provided for programming in 8080 assembly language.

- LOAD.COM — This programs makes ready-to-run programs out of assembled programs (programs with the extension .HEX).

- DUMP.COM — With this program a program (.COM) can be displayed on the screen in readable hexadecimal format.

- PIP.COM — PIP is a program to exchange data between different peripheral devices.

- ED.COM — The CP/M text editor. This program is useful for creating text, assembly language source programs and so on.

- SYSGEN.COM PIP can only copy files, so SYSGEN is needed to write the individual BIOS tracks on the disk and thereby generate a new BIOS (see section 6.3).

- MOVCPM.COM This program fits the standard CP/M to your special type of computer (see section 6.3).

- SUBMIT.COM It often happens that the same input must be entered repeatedly. The SUBMIT command allows you to create a file of this input which it will enter at the appropriate time (such as setting certain start-up parameters).

- XSUB.COM This program also eases the work of entering repeatedly occurring commands. It is only combination with SUBMIT. It allows manual input, over the keyboard, during the operation of SUBMIT.

At this point we cannot deal with all of these programs in detail. We shall limit our presentation to a brief introduction to three of the most commonly used programs. More information can be obtained from the CP/M manual.

STAT

STAT is one of the more important CP/M programs. It might also be called the STATUS program because one can obtain a great deal of information about the condition of the entire system from it. STAT not only gives the remaining space on the diskette and the length of the files, but it can also alter the read/write pointer which indicates whether the disk can be written to and read from, or may only be read from. There is one limitation when using the Commodore 64 version of CP/M. The I/O byte is not implemented which means that the individual assignments cannot be changed. This should not be a problem in normal use, however.

An example of the STAT program:

```
A>STAT VAL:

TEMP R/O DISK: D:=R/O
SET INDICATOR: D:FILENAME.TYP $R/O $R/W $SYS $DIR
DISK STATUS  : DSK: D:DSK:
USER STATUS  : USR:
IOBYTE ASSIGN:

CON: = TTY: CRT: BAT: UC1:
RDR: = TTY: PTR: UR1: UR2:
PUN: = TTY: PTP: UP1: UP2:
LST: = TTY: CRT: LPT: UL1:
```

Here we can find out which values and in what form we can (theoretically) change, but as already mentioned, the values under IOBYTE cannot be changed on the Commodore 64.

If we want to place a write protect on a disk, for example, we enter STAT A:R/O. This write protect remains as long as the device is turned on. To find out the CP/M device assignments, enter STAT A:DEV:. On the 64 the result will be:

```
A>STAT DEV:
CON: IS TTY:
RDR: IS TTY:
PUN: IS TTY:
LST: IS TTY:
```

Changing any of these assignments will have no effect on the Commodore 64.

The information about the disk characteristics is also very informative. Here we can learn the disk capacity, what the construction looks like, and much more. The appropriate command is

```
A>STAT DSK:

     A: DRIVE CHARACTERISTICS
  1088: 128 BYTE RECORD CAPACITY
   136: KILOBYTE DRIVE CAPACITY
    64: 32 BYTE DIRECTORY ENTRIES
    64: CHECKED DIRECTORY ENTRIES
   128: RECORD/ EXTENT
     8: RECORD/ BLOCK
    34: SECTORS/ BLOCK
     2: RESERVED TRACKS
```

PIP

PIP is a universal program for copying files. Not only can it copy between different disk drives, it can also send data intended for the screen to the printer. This has the advantage that the programs themselves do not have to be changed in order to have them print out results on the printer. In order to copy a entire diskette, the form PIP B:=A:*.* is used. This command copies all the files from drive A to drive B. To send data to the printer, we would use a command such as PIP LST:=DUMP.ASM. Now, provided that the printer is connected, the entire program DUMP.ASM will be printed.

ED

The text editor allows input of text or programs which will be later compiled or assembled, such as FORTRAN, COBOL, or assembly language programs. Working with ED requires some practice; it will appear somewhat complicated to Commodore users but it will not take long to become familiar with this simple editor. An example:

```
A>ED TEST.TXT
NEW FILE
*I
THIS IS THE FIRST TEXT LINE
AND THIS IS THE SECOND
<CTRL>-Z                 press the CTRL and the Z key
                         at the same time
*E
A>
```

These commands write the two lines of text shown beneath the "*I" to a disk file called TEST.TXT. The "*" is the editor's command prompt. Entering the E command ends the editing session and saves the file to disk, returning you to CP/M and the A> prompt. This file can be listed by entering TYPE TEST.TXT. Other editing commands for changing and manipulating the file from within ED are also available; consult your CP/M manual for details.

6.3 Adapting standard CP/M software to the 64

What must be taken into consideration when adapting CP/M software to the hardware of the Commodore 64? First you must remember that the screen is only 40 columns across. Because most CP/M software is written for an 80 column screen, you will need an 80 column card of some sort. In addition, a large, fast disk drive would be useful for working with CP/M.

To adapt CP/M to a specific computer, the operating system has two programs at its disposal: MOVCPM and SYSGEN. MOVCPM sets up the operating system for a specific memory configuration. It is possible to make use of the maximum memory capacity this way. When starting up the CP/M system, the computer responds with the message 44K CP/M. It is possible, however, to use the entire memory for CP/M. We have mentioned before that RAM can occupy the entire address space of the Commodore 64.

The CP/M system can be copied onto your own diskettes with SYSGEN, allowing you to create "boot" diskettes, disk which you can use to initialize or "boot-up" the CP/M system.

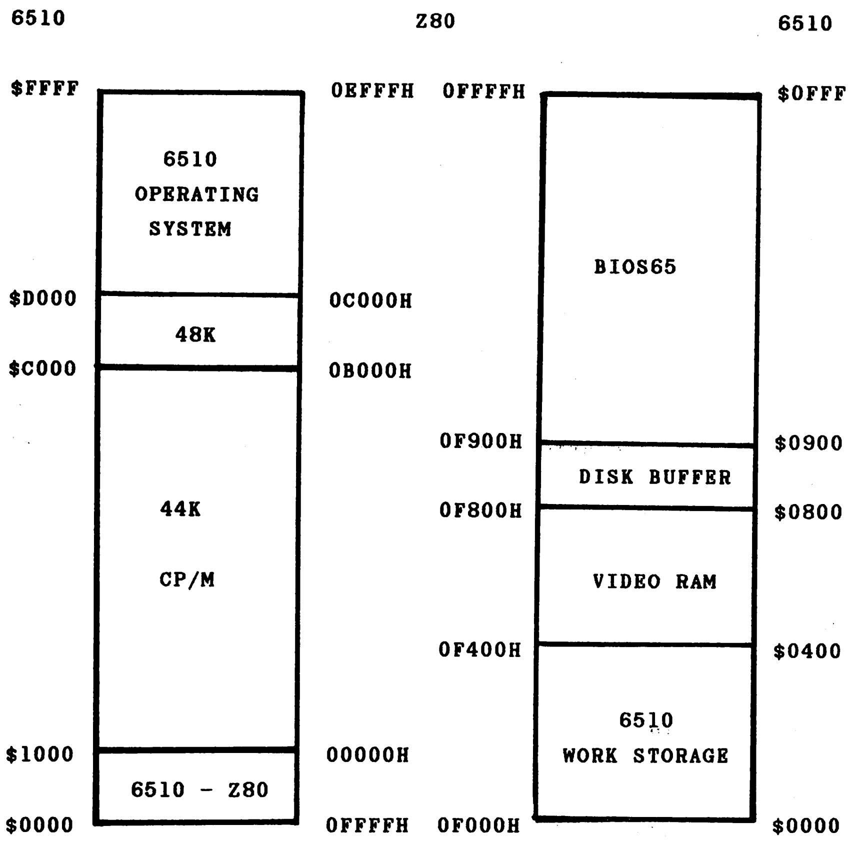
The problem of transferring existing CP/M programs to the VIC-1541 disk format is one the greatest obstacles to using CP/M on the Commodore 64. In section 6.9 we will present a method for transferring standard Commodore files (DOS 2.6 files) to CP/M. This can be used, together with an RS-232 serial interface and a terminal program which allows files to be down-loaded to the 64, to down-load programs from another CP/M computer for which software is available in the appropriate format. The connection between the two may be made directly using something called a null-modem cable or over the phone lines via modems. Once the program is saved as a DOS 2.6 file, it can be transferred to CP/M as described in section 6.9.

6.4 The memory management of the Z-80 processor

The Z-80 processor on the CP/M card can address the entire 64K bytes of the Commodore 64. Since the Z-80 requires address zero as the reset address (the address at which execution will start upon reset or power-up), and this address is assigned as the processor port of the 6510, the addressing of the memory through the Z-80 microprocessor is handled differently than the addressing through the 6510. The CP/M card contains the hardware to create an offset when addressing the memory through the Z-80. The offset is equal to $1000 or 4096. This results in the following situation: When the Z-80 wants address zero, the hardware manipulations of the address lines result in an address which is equivalent to address $1000 for the 6510. To calculate the corresponding Z-80 address from a 6510 address, simply subtract $1000. Alternatively, you can also add $F000 and ignore the overflow. Through this trick, an area of 4K bytes from address 0 to $0FFF on the 6510 is placed at the end of the address range of the Z-80 ($F000-$FFFF). This memory area contains the zero page, the stack, and the scratch pad memory of the 6510 as well as the video RAM. The other 2K from $800 to $FFF is used by the CP/M card to transmit data between the 6510 and the Z-80, as well as program storage for the 6510 input/output routines. The Z-80 delegates all of the input/output to the 6510

Label	6510	Z80	Description
HSTBUF	$0800	0F800H	256-byte disk buffer
CMD	$0900	0F900H	command register for the 6510
DATA	$0901	0F901H	data register

SECTOR	$0902	0F902H	sector register
TRACK	$0903	0F903H	track register
DISKNO	$0904	0F904H	register for drive number
KEYCHAR	$0905	0F905H	number of the pressed key
MODESW	$DE00	0CE00H	switch for 6510/Z80
IOTYPE	$0CFF	0FCFFH	I/O configuration



6.5 Disk management under CP/M

A diskette is divided into a number of concentric tracks which are further divided into sectors. These divisions are set up as follows on the 1541:

Track	Sectors
1-17	0-20
18-24	0-18
25-30	0-17
31-35	0-16

A total of 683 sectors (blocks) are available. Track 18 is used to store the directory, so 664 blocks are available for file storage.

Under CP/M, the first two tracks are reserved for the CP/M operating system itself; the other tracks are free for data and program storage. Because disk management under CP/M cannot make use of a variable number of sectors, only the sectors from 0 to 16 are used. In effect you have 32 tracks each containing 17 sectors, for a total of 574 256-byte blocks or 143K bytes of storage. In addition, the CP/M directory requires some space (64 entries or 32 bytes each, 2K bytes). This is stored in the CP/M BIOS (Basic Input/Output System) disk parameter block and can be adapted by the user to his disk capacity. Track 18 (which contains the Commodore directory) is not used by CP/M.

Track 1, sector 0 of the operating system disk contains the loader program "CPM." The "BIOS 65" is contained in track 1, sectors 1 through 5, which contains the I/O routines for the 6510 as well as the cold-start loaders for

CP/M. The program "CPM" loads these three sectors at addresses $0A00 to $0EFF. From there the block from $0E00 to $0EFF is transferred to address $1000 to $10FF. This is Z-80 address zero, at which the cold-start loader will be transferred. Finally, the 6510 switches itself off while switching the Z-80 on. The Z-80 begins executing the program at address zero, which loads the CP/M operating system from disk. The CCP and the BDOS (Command Control Processor and Basic Disk Operating System) occupy 22 sectors on tracks one and two, from track 1 sector 6 to track 2 sector 10. These sectors are also loaded in on a warm start with control-C; a star appears on the screen for each sector loaded in. The BIOS, which is loaded in only on a cold start, occupies five sectors from track 2 sector 11 to sector 16. The directory occupies sectors 0 through 7 on track three.

With the CP/M utility program COPY, only the necessary sectors are copied, depending on the option selected. For example, selecting "system tracks only" copies only sectors 1 and 2 as well as 18 and track 3 for the CP/M directory.

If you want to connect a different disk drive, using the IEEE bus for example, you must know the track and sector layout of the drive. No adaptation is necessary for the 4040 drive because it is completely compatible with the 1541. To make use of the greater storage capacity of the 8050 or 8250 dual drives, it is necessary to make some changes to the "disk parameter block" of the BIOS. There are the values for the sectors per track (23 for the 8050) and the disk capacity. In addition, the tracks 38 and 39 must be set aside instead of track 18 because the directory is stored there on the 8050. These changes must also be made in the COPY program.

6.6 The interaction between the 6510 and the Z-80

When you work with CP/M on the Commodore 64, the two microprocessors share the work. While the Z-80 serves the actual CP/M, the 6510 is brought into play to handle the input/output operations since the Commodore 64 already has these routines in its kernal ROM. The Z-80 delegates the following tasks to the 6510:

Command number	Operation
0	read a sector from the disk
1	write a sector to the disk
2	read the keyboard
3	display a character on the screen
4	get the printer status
5	output a character to the printer
6	format the diskette
7-9	reserved for future expansion, such as serial I/O

The two processors cannot operate simultaneously. The address \$DE00 of the 6510 (0CE00H on the Z-80) is used to switch between the two. When the Z-80 wants the 6510 to execute an I/O function, it sends the 6510 the number of the desired code from the above table and switches itself off while switching the 6510 on. It does this by writing a "1" to address 0CE00H. The 6510 fetches the command code, executes the appropriate command, and switches over to the Z-80 by writing a "0" to address \$DE00. The Z-80 can then continue with its program at the point where it had passed control to the 6510. Because of certain hardware requirements, the first command executed after switching from the 6510 to the Z-80 must be a NOP (No OPeration).

Six memory locations at address $900 (0F900H for the Z-80) are used to transmit parameters.

6510	Z-80	Parameter
$0900	0F900H	command number
$0901	0F901H	data for input or output
$0902	0F902H	sector number
$0903	0F903H	track number
$0904	0F904H	disk number
$0905	0F905H	key number

The memory from $800 to $8FF (0F800H to 0F8FFH) is used as a buffer to hold a sector to be written to the disk or one which has just been read from the diskette. Reading and writing disk sectors is performed with the direct access Block-Read and Block-Write commands.

The keyboard polling yields only the number of the depressed key. The assignment of an ASCII value to a key happens in the BIOS using a 256-byte table at address $D00 (0FD00H). A table at address $C00 (0FC00H) contains the addresses of the character strings assigned to the functions keys. The definitions themselves start at address $C10 (0FC10H) and may consist of up to 16 characters each. These assignments may be changed with the program CONFIG.

6.7 Implementing your own Input/Output routines in BIOS

The CP/M BIOS contains two routines called PUNCH and READER, which are not used in the Commodore 64 version of CP/M. The READER routine currently returns a control-Z, the marker for the end of the file. We can use it and the PUNCH vector for our own purposes. The PUNCH routine could be used to drive a printer with a Centronics-type parallel interface, for instance, as described in section 7.1. The driver routine can be formulated in 6510 code, and we can use the command codes 7 and 8 which branch to locations $E00 and $F00, respectively, to call our drivers. The call would look something like this:

```
PUNCH: ;output character to PUN: Centronics printer
       MOV    A,C       ;character in accumulator
       STA    DATA      ;into transfer register
       MVI    A,7       ;code for our routine
       STA    CMD
       CALL   IO6510    ;call 6510 routine
       RET
```

Our 6510 driver must be located at address $E00; we have 256 bytes available to us. Since the routine need only handlé outputting the data to the port and the handshaking, this will be plenty of room.

The READER routine can be implemented similarly.

```
READER: ;get character from RDR:
        MVI     A,8      ;code for our routine
        STA     CMD
        CALL    IO6510   ;call 6510 routine
        LDA     DATA     ;get character
        ANI     7FH      ;erase parity bit
        RET
```

The command code 8 expects the input routine to be at address $F00; here too we have 256 bytes available to us. You can define your favorite input device as RDR:, such as the cassette recorder, another disk drive, or an interface to transfer data from other computers. The READER routine expects text data in standard ASCII format. The end-of-file is indicated by a control-Z, as usual.

6.8 Transferring CP/M programs and data to and from Commodore BASIC

When one works with CP/M on the Commodore 64, one normally does not have the ability to later use programs or data in the normal BASIC mode of the 64. These files are only accessible in the CP/M mode. It is possible to transfer files, however, with a small change to the BIOS.

In CP/M you can send data to the printer. This is done in BIOS65 with the appropriate OPEN and PRINT# commands. At this point we can go in and simply change the device number on the OPEN command. If we set the number to one, all the data intended for the printer will be sent to the cassette recorder instead. We must enter one as the secondary address a well, so that the tape file will be opened for writing. We can make this change using the CP/M program DDT. Enter the following commands to make the changes (your input is underlined):

```
DDT
-SFAC7
FAC7 07 01
FAC8 20 .
-SFADD
FADD 00 01
FADE A9 .
-SFAE6
FAE6 04 01
FAE7 20 .
-^C
```

Now when you output something with ^P or PIP LST:=, it will not be sent to the printer but to the cassette drive. The first time, the message "press record & play on tape" will appear and the screen will go dark. Once the data has been sent, press control C and then press STOP and RESTORE together during the warm start. The computer will respond with "ready." in the Commodore mode. Now you must close the tape file with CLOSE 4, and after you have the turned the computer off and back on again, you can read the tape file in:

```
100 OPEN 1
110 GET#1, A$
...
200 IF ST<>64 THEN 110
210 CLOSE 1
```

This program gets the data character by character; you can then save it to a disk file or do whatever you like with it.

If you want to transfer data from CP/M often, you can use the editor (ED) and the assembler (ASM) to create a small program which makes the changes for you so that you do not have to use DDT. First create the following program using ED:

```
ORG 100H
MVI A,1
STA 0FAC7H
STA 0FADFH
STA 0FAE6H
JMP 0
```

and assemble it

```
ASM TAPE.AAX
```

Then make a .COM file out of it

```
LOAD TAPE
```

Now you can make the changes by simply typing the program name TAPE from CP/M.

It is also possible to transfer data the other way, from the Commodore BASIC mode to CP/M. To transfer a file, is must be saved as a program file with load address $1100. This is equivalent to the Z-80 address 100H, the start of the Transient Program Area (TPA). The following program will save a file as a program with a load address of $1100:

```
100 INPUT "NAME OF THE FILE ";N$
110 OPEN 2,8,2, N$
120 OPEN 1,8,1, N$+".CPM": REM OPEN PROGRAM FILE
130 PRINT#1, CHR$(0)CHR$(17); : REM START ADDRESS $1100
140 GET#2, A$: IF ST=64 THEN CLOSE 1:CLOSE 2: END
150 PRINT#1, A$;: GOTO 140
```

Before we can load the program, we must know its length. We can find this by loading the directory.

```
...
25   "NAME.CPM"        PRG
...
```

Remember the number 25. Now we load the program file.

```
LOAD "NAME.CPM",8,1
```

Insert the CP/M diskette and enter CP/M as usual. When CP/M is loaded, we can save the file under CP/M.

```
SAVE 25 NAME.TXT
```

Here the number 25 gives the number of 256-byte blocks to be saved, but is identical to the number of blocks given in the Commodore directory. There may be a problems with upper/lower case reversal when transferring text files. If this is the case, the conversion to standard ASCII can be made to A$ in line 140.

Chapter 7: Interface and expansion options

7.1 The USER port: An interface for a Centronics printer

The Commodore 64 has an interface which is not normally used by the operating system, and which is available for your own devices. This interface consists of an 8-bit port and two handshake lines. The 8-bit port can be used for input as well as output; each bit may be switched independently.

This interface is ideally suited for implementing a printer interface. Here in short is the procedure:

The 8 bits of a byte are sent in parallel over eight data lines. To insure that no data are lost during the transmission, two so-called "handshake lines" are used. Before the computer sends a data byte to the printer, it checks the BUSY line to see if the printer is ready to receive the data. If the BUSY line is high, the printer is not ready and the computer must wait. When the printer is ready, the computer sends the data over the port and signals the printer by means of the STROBE line that it is sending the data. The printer accepts the data and sets the BUSY line high until it is ready to receive another character. Now the next byte can be transmitted, and so on. This process ensures that the printer actually receives each byte sent by the computer.

In order to be able to use the PRINT# command to send data to the printer, the software in the operating system must be modified. There is also one additional problem:

Most of the printers with a Centronics-type interface use the standard ASCII character set, which is different from the Commodore 64's character set. We must convert the codes used by the computer to the equivalent codes used by the printer. In addition, it is also necessary to be able to send data to the printer exactly as the computer sends it. This is required when doing things like graphics on the printer.

To solve this problem, we have written the driver program to accept two options. If no secondary address is given along with the OPEN command, the data will be converted from the Commodore codes to the appropriate ASCII codes. If a secondary address of one is given, the data will be sent to the printer without alteration. The device address 2 was chosen. This address is normally used for the RS232 interface, but since this interface cannot be used in conjunction with our interface (it also uses the USER port), this presents no problems.

To send a program listing over this interface, you would enter the following commands:

```
OPEN 1,2 : CMD 1 : LIST
```

After the cursor reappears, enter

```
PRINT#1 : CLOSE 1
```

to return the CMD mode to normal and close the channel. If you want to transmit graphics data or printer commands, the following would be used:

```
OPEN 1,2,1
PRINT# 1, ...
CLOSE 1
```

For the hardware portion of the interface, all that is needed is a cable with a USER port socket on one end and a Centronics socket on the other. The pin layout of the cable is given at the end of the assembly listing. When connecting the printer, attach the cable between the printer and computer, turn the computer on, and then turn the printer on. Load the machine language program and initialize it with SYS 12*4096.

```
0001 C000                                         ;CENTRONICS I
NTERFACE DRIVER FOR CBM 64
0002 C000                                         ;PRINTER CONN
ECTED TO USER PORT
0003 C000                                         ;
0004 C000                                         ;DEFINITION O
F THE I/O VECTORS
0005 C000                                         ;
0006 C000            OPENV  EQU $31A
0007 C000            CLOSEV EQU $31C
0008 C000            CHKINV EQU $31E
0009 C000            CHKOTV EQU $320
0010 C000            BSOUTV EQU $326
0011 C000            XREG   EQU $97               ;STORAGE FOR
REGISTER
0012 C000                                         ;
0013 C000                                         ;PORT DEFINIT
IONS
0014 C000                                         ;
0015 C000            PORTA  EQU $DD00             ;CIA2 PORT
0016 C000            PORTB  EQU $DD01
0017 C000            DDRA   EQU $DD02             ;DATA DIRECTI
ON
0018 C000            DDRB   EQU $DD03
0019 C000            ICR    EQU $DD0D             ;INTERRUPT CO
NTROL REGISTER
0020 C000            LF     EQU $B8
0021 C000            SA     EQU $B9
0022 C000            FA     EQU $BA
0023 C000            NMBFLS EQU $98
0024 C000            LFTAB  EQU $259
0025 C000            FATAB  EQU $263
0026 C000            SATAB  EQU $26D
```

```
C000              SRCHFL EQU $F30F
C000                                         ;
C000                                         ;INITIALIZATI
ON
C000                                         ;
C000                     ORG $C000
C000 A9 58               LDA #<OPEN
C002 A0 C0               LDY #>OPEN
C004 8D 1A 03            STA OPENV
C007 8C 1B 03            STY OPENV+1
C00A A9 8B               LDA #<CLOSE
C00C A0 C0               LDY #>CLOSE
C00E 8D 1C 03            STA CLOSEV
C011 8C 1D 03            STY CLOSEV+1
C014 A9 A3               LDA #<CHKIN
C016 A0 C0               LDY #>CHKIN
C018 8D 1E 03            STA CHKINV
C01B 8C 1F 03            STY CHKINV+1
C01E A9 BA               LDA #<CHKOUT
C020 A0 C0               LDY #>CHKOUT
C022 8D 20 03            STA CHKOTV
C025 8C 21 03            STY CHKOTV+1
C028 A9 D1               LDA #<BSOUT
C02A A0 C0               LDY #>BSOUT
C02C 8D 26 03            STA BSOUTV
C02F 8C 27 03            STY BSOUTV+1
C032 A9 FF               LDA ##$FF
C034 8D 03 DD            STA DDRB            ;PORT B AS OU
TPUT
C037 AD 02 DD            LDA DDRA
C03A 09 04               ORA ##$04
C03C 8D 02 DD            STA DDRA            ;PA2 AS OUTPU
T
C03F 60                  RTS
C040                                         ;
C040                                         ;OUTPUT WITH
HANDSHAKE
C040                                         ;DATA TO PORT
 B
C040                                         ;STROBE ON PA
2
C040                                         ;BUSY OVER FL
AG TO ICR
C040                                         ;
C040 8D 01 DD     OUTPUT STA PORTB           ;OUTPUT DATA
C043 A9 10               LDA ##$10           ;MASK FOR 'FL
AG' BIT
C045 2C 0D DD     TSTBSY BIT ICR
C048 F0 FB               BEQ TSTBSY
C04A AD 00 DD            LDA PORTA
C04D 29 FB               AND ##$FB           ;ERASE STROBE
C04F 8D 00 DD            STA PORTA
C052 09 04               ORA ##$04           ;SET STROBE
C054 8D 00 DD            STA PORTA
```

```
C057 60                 RTS
C058                                            ;
C058 A6 B8     OPEN     LDX LF                  ;LOGICAL FILE
 NUMBER
C05A F0 05              BEQ OPNERR
C05C 20 0F F3           JSR SRCHFL              ;SEARCH FOR F
ILE NUMBER
C05F D0 03              BNE LB1
C061 4C FE F6  OPNERR   JMP $F6FE               ;'FILE OPEN E
RROR'
C064 A6 98     LB1      LDX NMBFLS              ;NUMBER OF OP
EN FILES
C066 E0 0A              CPX #10
C068 90 03              BCC LB2
C06A 4C FB F6           JMP $F6FB               ;'TOO MANY FI
LES ERROR'
C06D E6 98     LB2      INC NMBFLS
C06F A5 B8              LDA LF
C071 9D 59 02           STA LFTAB,X
C074 A5 B9              LDA SA
C076 09 60              ORA #$60
C078 9D 6D 02           STA SATAB,X
C07B A5 BA              LDA FA
C07D 9D 63 02           STA FATAB,X
C080 C9 02              CMP #2
C082 D0 02              BNE LB3
C084 18                 CLC
C085 60                 RTS                     ;DONE
C086 C9 00     LB3      CMP #0
C088 4C 77 F3           JMP $F377
C08B                                            ;
C08B 20 14 F3  CLOSE    JSR $F314               ;SEARCH FOR L
OGICAL FILE NUMBER
C08E F0 02              BEQ LB4
C090 18                 CLC
C091 60                 RTS                     ;DONE
C092 20 1F F3  LB4      JSR $F31F               ;SET FILE PAR
AMETERS
C095 8A                 TXA
C096 48                 PHA
C097 A5 BA              LDA FA
C099 C9 02              CMP #2
C09B F0 03              BEQ LB5
C09D 4C 9D F2           JMP $F29D               ;NORMAL CONTI
NUE
C0A0 4C F1 F2  LB5      JMP $F2F1               ;ERASE ENTRY
IN TABLE
C0A3                                            ;
C0A3 20 0F F3  CHKIN    JSR SRCHFL              ;SEARCH FOR F
ILE NUMBER
C0A6 F0 03              BEQ LB6
C0A8 4C 01 F7           JMP $F701               ;'FILE NOT OP
EN ERROR'
C0AB 20 1F F3  LB6      JSR $F31F               ;SET FILE PAR
AMETERS
```

```
C0AE A5 BA              LDA FA
C0B0 C9 02              CMP #2
C0B2 D0 03              BNE LB7
C0B4 4C 0A F7           JMP $F70A          ;'NOT INPUT F
ILE ERROR'
C0B7 4C 19 F2   LB7     JMP $F219
C0BA 20 0F F3   CHKOUT  JSR SRCHFL         ;SEARCH FOR F
ILE NUMBER
C0BD F0 03              BEQ LB8
C0BF 4C 01 F7           JMP $F701          ;'FILE NOT OP
EN ERROR'
C0C2 20 1F F3   LB8     JSR $F31F          ;SET FILE PAR
AMETERS
C0C5 A5 BA              LDA FA
C0C7 C9 02              CMP #2
C0C9 D0 03              BNE LB9
C0CB 4C 75 F2           JMP $F275
C0CE 4C 5B F2   LB9     JMP $F25B
C0D1                                       ;
C0D1 48         BSOUT   PHA
C0D2 A5 9A              LDA $9A            ;OUTPUT DEVIC
E
C0D4 C9 02              CMP #2
C0D6 F0 03              BEQ LB10
C0D8 4C CD F1           JMP $F1CD          ;NORMAL CONTI
NUE
C0DB A5 B9      LB10    LDA SA             ;SECONDARY AD
DRESS
C0DD 29 0F              AND #$0F
C0DF D0 0A              BNE OUT            ;NOT EQUAL TO
 ZERO
C0E1 86 97              STX XREG
C0E3 68                 PLA
C0E4 AA                 TAX
C0E5 BD F3 C0           LDA TABLE,X        ;GET CODE FRO
M TABLE
C0E8 A6 97              LDX XREG
C0EA 24                 BYT $24
C0EB 68         OUT     PLA
C0EC 48                 PHA
C0ED 20 40 C0           JSR OUTPUT         ;OUTPUT CHARA
CTER
C0F0 68                 PLA
C0F1 18                 CLC
C0F2 60                 RTS
C0F3 00 01 02   TABLE   BYT $00,$01,$02
C0F6 03 04 05           BYT $03,$04,$05
C0F9 06 07 08           BYT $06,$07,$08
C0FC 09 0A 0B           BYT $09,$0A,$0B
C0FF 0C 0D 0E           BYT $0C,$0D,$0E
C102 0F 10 11           BYT $0F,$10,$11
C105 12 13 14           BYT $12,$13,$14
C108 15 16 17           BYT $15,$16,$17
C10B 18 19 1A           BYT $18,$19,$1A
```

```
C10E 1B 1C 1D       BYT $1B,$1C,$1D
C111 1E 1F 20       BYT $1E,$1F,$20
C114 21 22 23       BYT $21,$22,$23
C117 24 25 26       BYT $24,$25,$26
C11A 27 28 29       BYT $27,$28,$29
C11D 2A 2B 2C       BYT $2A,$2B,$2C
C120 2D 2E 2F       BYT $2D,$2E,$2F
C123 30 31 32       BYT $30,$31,$32
C126 33 34 35       BYT $33,$34,$35
C129 36 37 38       BYT $36,$37,$38
C12C 39 3A 3B       BYT $39,$3A,$3B
C12F 3C 3D 3E       BYT $3C,$3D,$3E
C132 3F 40 61       BYT $3F,$40,$61
C135 62 63 64       BYT $62,$63,$64
C138 65 66 67       BYT $65,$66,$67
C13B 68 69 6A       BYT $68,$69,$6A
C13E 6B 6C 6D       BYT $6B,$6C,$6D
C141 6E 6F 70       BYT $6E,$6F,$70
C144 71 72 73       BYT $71,$72,$73
C147 74 75 76       BYT $74,$75,$76
C14A 77 78 79       BYT $77,$78,$79
C14D 7A 7B 7C       BYT $7A,$7B,$7C
C150 7D 7E 5F       BYT $7D,$7E,$5F
C153 60 61 62       BYT $60,$61,$62
C156 63 64 65       BYT $63,$64,$65
C159 66 67 68       BYT $66,$67,$68
C15C 69 6A 6B       BYT $69,$6A,$6B
C15F 6C 6D 6E       BYT $6C,$6D,$6E
C162 6F 70 71       BYT $6F,$70,$71
C165 72 73 74       BYT $72,$73,$74
C168 75 76 77       BYT $75,$76,$77
C16B 78 79 7A       BYT $78,$79,$7A
C16E 7B 7C 7D       BYT $7B,$7C,$7D
C171 7E 7F 80       BYT $7E,$7F,$80
C174 81 82 83       BYT $81,$82,$83
C177 84 85 86       BYT $84,$85,$86
C17A 87 88 89       BYT $87,$88,$89
C17D 8A 8B 8C       BYT $8A,$8B,$8C
C180 8D 8E 8F       BYT $8D,$8E,$8F
C183 90 91 92       BYT $90,$91,$92
C186 93 94 95       BYT $93,$94,$95
C189 96 97 98       BYT $96,$97,$98
C18C 99 9A 9B       BYT $99,$9A,$9B
C18F 9C 9D 9E       BYT $9C,$9D,$9E
C192 9F A0 A1       BYT $9F,$A0,$A1
C195 A2 A3 A4       BYT $A2,$A3,$A4
C198 A5 A6 A7       BYT $A5,$A6,$A7
C19B A8 A9 AA       BYT $A8,$A9,$AA
C19E AB AC AD       BYT $AB,$AC,$AD
C1A1 AE AF B0       BYT $AE,$AF,$B0
C1A4 B1 B2 B3       BYT $B1,$B2,$B3
C1A7 B4 B5 B6       BYT $B4,$B5,$B6
C1AA B7 B8 B9       BYT $B7,$B8,$B9
C1AD BA BB BC       BYT $BA,$BB,$BC
C1B0 BD BE BF       BYT $BD,$BE,$BF
```

```
0215 C1B3 C0 41 42          BYT $C0,$41,$42
0216 C1B6 43 44 45          BYT $43,$44,$45
0217 C1B9 46 47 48          BYT $46,$47,$48
0218 C1BC 49 4A 4B          BYT $49,$4A,$4B
0219 C1BF 4C 4D 4E          BYT $4C,$4D,$4E
0220 C1C2 4F 50 51          BYT $4F,$50,$51
0221 C1C5 52 53 54          BYT $52,$53,$54
0222 C1C8 55 56 57          BYT $55,$56,$57
0223 C1CB 58 59 5A          BYT $58,$59,$5A
0224 C1CE 5B 5C 5D          BYT $5B,$5C,$5D
0225 C1D1 DE DF E0          BYT $DE,$DF,$E0
0226 C1D4 E1 E2 E3          BYT $E1,$E2,$E3
0227 C1D7 E4 E5 E6          BYT $E4,$E5,$E6
0228 C1DA E7 E8 E9          BYT $E7,$E8,$E9
0229 C1DD EA EB EC          BYT $EA,$EB,$EC
0230 C1E0 ED EE EF          BYT $ED,$EE,$EF
0231 C1E3 F0 F1 F2          BYT $F0,$F1,$F2
0232 C1E6 F3 F4 F5          BYT $F3,$F4,$F5
0233 C1E9 F6 F7 F8          BYT $F6,$F7,$F8
0234 C1EC F9 FA FB          BYT $F9,$FA,$FB
0235 C1EF FC FD FE          BYT $FC,$FD,$FE
0236 C1F2 FF                BYT $FF
```

```
100 FOR I = 49152 TO 49650
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 169, 88,160,192,141, 26,  3,140, 27,  3,169,139
130 DATA 160,192,141, 28,  3,140, 29,  3,169,163,160,192
140 DATA 141, 30,  3,140, 31,  3,169,186,160,192,141, 32
150 DATA   3,140, 33,  3,169,209,160,192,141, 38,  3,140
160 DATA  39,  3,169,255,141,  3,221,173,  2,221,  9,  4
170 DATA 141,  2,221, 96,141,  1,221,169, 16, 44, 13,221
180 DATA 240,251,173,  0,221, 41,251,141,  0,221,  9,  4
190 DATA 141,  0,221, 96,166,184,240,  5, 32, 15,243,208
200 DATA   3, 76,254,246,166,152,224, 10,144,  3, 76,251
210 DATA 246,230,152,165,184,157, 89,  2,165,185,  9, 96
220 DATA 157,109,  2,165,186,157, 99,  2,201,  2,208,  2
230 DATA  24, 96,201,  0, 76,119,243, 32, 20,243,240,  2
240 DATA  24, 96, 32, 31,243,138, 72,165,186,201,  2,240
250 DATA   3, 76,157,242, 76,241,242, 32, 15,243,240,  3
260 DATA  76,  1,247, 32, 31,243,165,186,201,  2,208,  3
270 DATA  76, 10,247, 76, 25,242, 32, 15,243,240,  3, 76
280 DATA   1,247, 32, 31,243,165,186,201,  2,208,  3, 76
290 DATA 117,242, 76, 91,242, 72,165,154,201,  2,240,  3
300 DATA  76,205,241,165,185, 41, 15,208, 10,134,151,104
310 DATA 170,189,243,192,166,151, 36,104, 72, 32, 64,192
320 DATA 104, 24, 96,  0,  1,  2,  3,  4,  5,  6,  7,  8
330 DATA   9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
340 DATA  21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
350 DATA  33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44
360 DATA  45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56
370 DATA  57, 58, 59, 60, 61, 62, 63, 64, 97, 98, 99,100
380 DATA 101,102,103,104,105,106,107,108,109,110,111,112
390 DATA 113,114,115,116,117,118,119,120,121,122,123,124
400 DATA 125,126, 95, 96, 97, 98, 99,100,101,102,103,104
410 DATA 105,106,107,108,109,110,111,112,113,114,115,116
420 DATA 117,118,119,120,121,122,123,124,125,126,127,128
```

```
430 DATA 129,130,131,132,133,134,135,136,137,138,139,140
440 DATA 141,142,143,144,145,146,147,148,149,150,151,152
450 DATA 153,154,155,156,157,158,159,160,161,162,163,164
460 DATA 165,166,167,168,169,170,171,172,173,174,175,176
470 DATA 177,178,179,180,181,182,183,184,185,186,187,188
480 DATA 189,190,191,192, 65, 66, 67, 68, 69, 70, 71, 72
490 DATA  73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84
500 DATA  85, 86, 87, 88, 89, 90, 91, 92, 93,222,223,224
510 DATA 225,226,227,228,229,230,231,232,233,234,235,236
520 DATA 237,238,239,240,241,242,243,244,245,246,247,248
530 DATA 249,250,251,252,253,254,255
540 IF S <> 58534 THEN PRINT "ERROR IN DATA!!" : END
550 PRINT "OK"
```

The cable connecting the printer to the User port has the following pin layout:

USER PORT	-	CENTRONICS
A	GND	16
B	FLAG-BUSY	11
C	D0	2
D	D1	3
E	D2	4
F	D3	5
H	D4	6
J	D5	7
K	D6	8
L	D7	9
M	PA2-STROBE	1

7.2 Transferring data between computers using the USER port

Imagine the following: You own, in addition to your Commodore 64, a CBM 8032. Wouldn't it be nice to be able to directly transfer data from the 8032 to your 64 where you can work with it in color? Or maybe you like to be able to send information from the 64 to the 8032, where you can see it in 80 columns. You could do this with a cassette, assuming you have one, but this is a tedious and bothersome process.

We have chosen this example to illustrate the use of the Commodore 64's USER port, and we have written a small program which allows the 64 to both send and receive data. In our case, the device to which we will be sending data (or from which we will be receiving it) is a CBM 8032. It is also possible to use the same procedure for communicating with other computers which have an interface similar to the user port.

The programs (one for the Commodore 64 and one for the CBM 8032) which will be given shortly naturally require the appropriate connection between the two computers. The pin assignments and necessary connections can be found following the listing. First, however, we will present a detailed account of the variables and memory locations used in each of the programs as well as a step-by-step description of the programs themselves.

The variables (both programs):

X ASCII value of a sent or received byte
TI operating system clock; counts in 1/60 second steps
D$ composite string of data received or to be sent

Memory locations used on the CBM 8032:

59457 Data register of the user port

59459 User port data direction register. As you may already know, the user port may be configured as either input or output. For this reason, we must specify the data direction of each bit.

59468 Bit 5 of this address controls the CB2 line of the user port. When sending, this line will indicate the validity of the data. When receiving, this lines serves as the acknowledgement signal, indicating that the data was received. These signals are required to ensure that no data are lost.

59469 Bit 1 gives the condition of the CA1 line of the user port. When sending data, this bit will be monitored for the acknowledgement signal of the receiving device, while when receiving, it is monitored as the data ready or valid signal.

Memory locations used on the Commodore 64:

56576 Bit 2 controls the PA2 line of the user port. This line is used in the same manner as the CB2 line on the CBM 8032 (see description of location 59468).

56577 User port data register.

56579 User port data direction register.

56589 Bit 4 reflects the condition of the FLAG line on the user port. Its use is the same as the CA1 line on the 8032 (see description of location 59469).

Program operation:

1000-1080 Send routine

1000 The data direction register is set to output.

1010 The length of the send loop is determined by the number of bytes to be sent.

1020 The individual bytes in D$, indexed by I, are written to the data register.

1030 The appropriate signal line is set to zero to indicate that the data on the data lines is valid.

1040 This loop waits until the receiver acknowledges reception of the data byte.

1050-1060 The data valid signal is set back to 1 and the next byte is sent.

1070-1080 The user port is returned to normal and the send routine ends.

2000-2090 Receive routine

2000 The first data byte is awaited.

2020-2030 The clock is set to zero and the next statements wait until either a data byte is received or two seconds have elapsed. If the latter is the case, it can be assumed that the data transmission is done. The time limit in line 2020 can be changed as desired. The 120 refers to 120 1/60ths of a second, a total of two seconds. To allow for a three second pause, the appropriate value would be 180.

2040-2050 The computer waits for the data valid signal. If it is received, the byte in X is added to D$.

2060-2080 As acknowledgement that the data byte was received, the corresponding line is set to zero and the loop returns to wait for the next character.

These programs consist only of two subroutines, one for sending data and one for receiving it. They should be inserted into your own programs. When you want to send characters, place them in D$ and GOSUB 1000. To receive data, call line 2000 (GOSUB 2000) and upon return, D$ will contain the characters received.

The first listing is for the CBM 8032 and the second is for the Commodore 64. They are identical in structure, although the addresses of the user ports are different in each case. The only other difference occurs in line 2010. In consideration for the different way in which the C64 kernal operates, a jump must be inserted.

8032 version

```
995 rem  subroutines for transferring data over the user
         port
996 rem  CBM 8032 for the 6522 at address 59456
997 rem
999 rem send a string
1000 poke 59459,255 : rem set data direction to output
1005 poke 59468,peek(59468) or 224 : rem cb2 high
1010 for i=1 to len(d$) : rem send loop for d$
1020 x=asc(mid$(d$,i,1)) : poke 59457,x : rem output data
1030 poke 59468,peek(59468) and 223 : rem cb2 low
1040 wait 59469,2 : rem wait until data received
1050 poke 59468,peek(59468) or 224 : rem cb2 high
1060 next
1070 poke 59457,0 : poke 59459,0 : rem reset port
1080 return
1996 rem
1997 rem receive data into d$
1998 rem
1999 rem
2000 wait 59469,2 : rem wait for start of data transmission
2010 d$="" : rem initialize d$
2020 ti$="000000"
2030 if ti>120 then 2090
2040 if (peek(59469) and 2)=0 then 2030 : rem wait for
                                              data byte
2050 x=peek(59457) : d$=d$+chr$(x)
2060 poke 59468,peek(59468) and 223 : rem cb2 low
2070 poke 59468,peek(59468) or 224 :
     rem cb2 high = receive confirmation
2080 goto 2020
2090 return
```

Commodore 64 version

```
995 rem  subroutines for transferring data over the user
         port
996 rem  CBM 64 version for 6526 at address 56576
998 rem
999 rem send a string
1000 poke 56579,255 : rem set data direction to output
1010 for i=1 to len(d$) : rem send loop for d$
1020 x=asc(mid$(d$,i,1)) : poke 56577,x : rem output data
1030 poke 56576,147 : rem pa2 low
1040 wait 56589,16 : rem wait until data received
1050 poke 56576,151: rem pa2 high
1060 next
1070 poke 56577,0 : poke 56579,0 : rem reset port
1080 return
1996 rem
1997 rem receive into d$
1998 rem
1999 rem
2000 wait 56589,16 : rem wait for start of transmission
2010 d$="" : goto 2050 : rem initialize d$
2020 ti$="000000"
2030 if ti>120 then 2090
2040 if (peek(56589) and 16)=0 then 2030 : rem wait for
                                                 data byte
2050 x=peek(56577) : d$=d$+chr$(x)
2060 poke 56576,147 : rem pa2 low
2070 poke 56576,151 : rem pa2 high = receive confirmation
2080 goto 2020
2090 return
```

A short example will clarify the use of these routines:

Assuming that you have loaded the appropriate routines into both computers, add the following line to the sender routine:

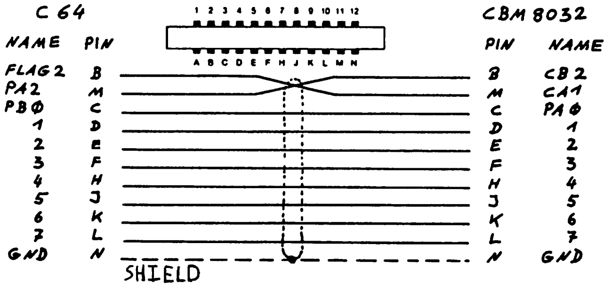
```
10 D$="test" : GOSUB 1000 : END
```

and the following to the receiver

```
10 GOSUB 2000 : PRINT D$ : END
```

Start both programs and watch what happens (assuming both computers are connected properly).

The diagram below shows the construction of the connecting cable. We recommend a 10-wire shielded cable. The shield is connected to the GND pin on both sides. It is best to limit the length of the cable to no more than 15 feet. If a longer cable is required, line drivers may have to be used to insure that no data is lost during transmission. Noise created by electric motors (washing machine, vacuum cleaner) or other large electrical devices can scramble the data being transmitted. You should have no problems at all if the length is kept under 10 feet. Ten feet should also be considered an absolute maximum when using an unshielded cable.



7.3 The CP/M cartridge on the expansion port: A case study

In this section, we describe how a clever piece of hardware can make optimal use of the expansion slot on the Commodore 64. For a better understanding of the material that we present, a knowledge of the material in the corresponding chapter of our book The Anatomy of the Commodore 64 is an advantage.

First of all, what it is the CP/M cartridge?

The CP/M cartridge is a module developed by Commodore which contains a Z-80 microprocessor and the necessary logic to communicate with the C64. The module makes it possible to use the popular CP/M operating system on the 64 and so gain access to the wide range of software available for CP/M.

The topic we wish to examine more closely is the technique of using two processors in the same computer. At the end of this section you will find a block diagram of the CP/M cartridge. Not all of the connections are shown in order to keep the diagram as simple as possible. The following discussion centers around this diagram and presents the function groups together with their designations. We have tried to make this discussion simple enough so that you need not be a hardware expert in order to understand it.

First, we present a description of the expansion port lines which play a role in this context:

CD0-7 System bus data lines.
These can be controlled by the 6510 within the 64 only as long as DMA=1 and BA=1.

We should make note of this condition, since it is necessary for further progress.

CA0-15 System bus address lines.
The above conditions apply to these lines as well.

I/O1 This line is low whenever any activities take place in the address range $DE00-$DEFF (56832-57087).

RES When this line is low (usually only when the computer is first turned on), all connected hardware devices are reset.

DMA This line is an input. Setting it to zero halts the 6510, leaving the system bus free for external control.

BA The 64's video controller uses this line to signal that it is accessing the memory (BA=0). During this time, the system bus may not be used by the 6510 or any other device.

S02 This is the system clock which coordinates all of the operations within the 64. In order to execute all activities in synchronization with the 64, the Z-80 in the CP/M module is also controlled by this clock.

We begin our description of the additional processes with the reset state, the condition of the device after being turned on. First we need an explanation of a line on the Z-80, BUSREQ. This signal has the same operation as the AEC (activated by DMA=0) on the 6510. If BUSREQ=0, the Z-80 ceases all activities and leaves its system bus free.

When the device is turned on, the RES line is set low for a short time, resetting the Z-80 and the FF flip-flop (Q=0, -Q=1). This has the effect of setting the output of the AND gate to zero, independent of BA. This in turn inhibits the A1, A2, and D buffers, preventing the Z-80 system bus from being externally controlled. BUSREQ is also held low, effectively inhibiting the Z-80 processor.

You can see now that the operation of the module depends on the condition of the flip-flop FF in combination with the signal BA (combined through the AND gate &). Only when FF is set (Q=1, -Q=0) and BA=1 does BUSREQ=1, allowing the Z-80 to operate. You can use your 64 as usual, provided you do not execute a certain command, namely POKE 56832,1.

As you can gather from the block diagram and the description of expansion port, this poke activates the line I/O1. Poking the value 1 sets our flip-flop FF and allows the Z-80 to run free, since BA=1 most of the time. At the same time, the 6510 is switched off and the computer will probably crash because there is no program in memory which will make any sense to the Z-80.

At this point we come to our next theme: Where must a program be so that the Z-80 can execute it? To find this out, we must dig a bit deeper. In contrast to the 6510, the Z-80 begins execution at location 0 after reset (RES=0). Here we have a conflict since the 6510 has its I/O port at location 0 and the following locations are the zero page, a section of memory absolutely required by the processor because the important system parameters are stored there. A Z-80 program simply cannot be stored at this point. On the other hand, we cannot change where the Z-80 will begin its execution.

The CP/M module solves this dilemma quite elegantly. If you take a look at the block diagram, you will find a function block which we have labeled ADD. This function block contains a four-bit full adder. The adder accepts two 4-bit words as input, adds them, and places the sum at its outputs. In our case, the adder is connected to the four highest-order bits of the address. One input is connected to the address lines of the Z-80 and the other is permanently set at 0001. The result is that the top four address bits are incremented by one. This has the net effect of increasing the total address by $1000 (4096) because the most significant digit of a two-byte address counts in 4K increments.

To return to our example, when the Z-80 outputs address zero in order to fetch the first command, it actually accesses address $1000 (4096). There, a program intended for the Z-80 can be placed without disturbing the operating system of the 6510. Using this scheme, a Z-80 address of $F000 (61440) corresponds to an effective address of 0, since the carry produced by the addition is ignored.

This procedure is essentially the same as the real operation of the module: After turning the computer on, a small start program is loaded into memory (at $1000 of course) and after setting the flip-flop FF, the Z-80 takes over and executes the program which loads the CP/M operating system. You should use this procedure when you want to execute Z-80 programs of your own. Simply place the program you have written at location $1000 (4096) and switch the cartridge on as described.

Since such a program is not an end in itself, but will have some output, whether to the screen or on the printer, a good knowledge of the Commodore 64's hardware is indispensable so that you can execute the appropriate functions from the Z-80 program. Remember that all addresses referenced from the module are offset by 4096. To send data to the user port, for example, you should use the address $CD01 (52481) in your Z-80 program because the user port is located at address $DD01 (56577) in the 6510 mode.

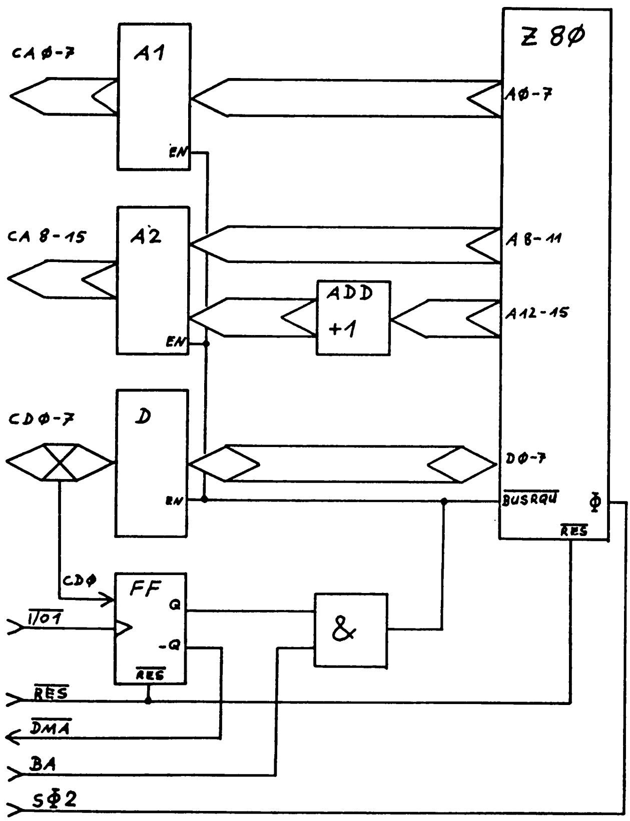
How does one return from the Z-80 mode? If the BASIC interpreter is loaded, you can reset FF by entering POKE 52736,0. This has the additional effect of setting BUSREQ to 0, halting the Z-80 while setting DMA to 1, whereupon the 6510 resumes execution at the point at which it left off.

The address must actually be 4096 less than the 6510 address since this value will be added back in by ADD. It is not recommended to proceed in this manner, however, since the 6510 will not find a program which it can run upon return since the memory contains programs intended for the Z-80, and the computer will crash.

BA is only a help signal which controls the traffic on the bus. It has a profound effect on the CP/M module, however, since the Z-80 halts execution whenever BA=0 or the output of FF is zero. If we take a closer look at the origin of BA, we will discover why this is so.

BA is a signal created by the video controller in the Commodore 64. Because the video controller must access the video RAM cyclically for refreshing the screen, the system bus must not be used by any other device. Normally this does not require halting the usual bus traffic; the video controller makes use of the "holes" in the microprocessor access cycles during which the processor is not using the system bus. There are exceptions, however, such as when the sprites are being displayed. Here these holes will not suffice since the memory must be accessed several times in succession. The video controller signals this condition by setting BA to zero and all other devices (including the Z-80 and the 6510) must give up the bus.

We have purposely kept the block diagram on the next page simple although the circuitry in the area of data buffer D is more complex than that shown. It is sufficient however to explain to explain the interaction between the Z-80 and the 6510.


CA Ø-7
A1
EN
Z 8Ø
AØ-7
CA 8-15
A2
A 8-11
ADD
+1
A12-15
EN
CD Ø-7
D
DØ-7
EN
BUSRQU
Φ
RES
CDØ
FF
Q
-Q
I/O1
&
RES
RES
DMA
BA
SΦ2

7.4 Synthesizer in stereo

If you use the synthesizer in your Commodore 64 often, you have probably wished for something better than the tinny sound of your TV speaker. With the help of a stereo receiver or amplifier we can produce considerably better sound. Because the stereo has two channels at its disposal, we must consider how to divide the single-channel output of the synthesizer between them. Unfortunately, the individual voices of the device do not have separate outputs, or we could make the division easily.

We have made certain allowances, however, and have divided the tone signal into two frequency ranges. The division occurs at 300 Hz. This splits the range of the synthesizer nearly in the middle as far as the ear is concerned, with three octaves below (down to 36 Hz) and four octaves above (up to 4800 Hz) 300 Hz.

This is accomplished with two double-T filters with an attenuation of 6dB/octave and a cut-off frequency of 300 Hz (low pass) and 3 kHz (high pass). You can change the cut-off frequencies as desired by using different capacitors, but you should leave the values of the resistors as they are since they were calculated to match the impedance of the connected device.

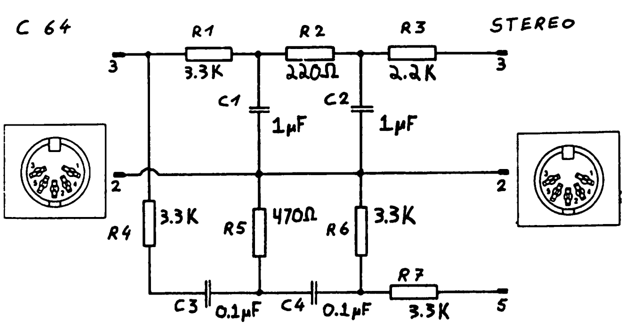
Given the cut-off frequency, the required value of the capacitor can be found with the formula C=1/(3300*F). If you have some capacitors already and want to know what cut-off frequency they would give, use the formula F=1/(3300*C). The values that we have chosen are optimized for this project based on numerous measurements and listening tests.

If an attenuation of 3dB/octave is good enough for you, the components R2, C2, C4, and R6 can be eliminated. This will result in a greater acoustical overlap between the two speakers. As you see, the filter circuit is extremely simple. It can be constructed on a piece of perfboard.

We now want to present a program which will produce a "sweep" using the triangle wave. This will allow you to clearly hear how the tone moves from one speaker to the other. We have chosen the triangle wave because it is relatively free of overtones and will allow the effect to noticed better. With more complex sounds, rich in overtones, such as the sawtooth, the overtones will appear on one channel while the fundamental wave will be heard on the other, provided that the fundamental does not exceed 300 Hz.

```
10 V1=54272
20 V2=54279
30 V3=54286
60 RS=54295
70 PL=54296
80 POKE V1+4,0 : POKE V2+4,0 : POKE V3+4,0
100 A=9 : D=9 : S=9 : R=9 : H=30
110 POKE RS,0 : POKE PL,15
120 POKE V3+5,16*A+D : POKE V3+6,16*S+R
130 POKE V3+4,17
140 FOR I=0 to H : POKE V3+1,PEEK(54300) : NEXT I
150 POKE V3+4,16
160 FOR I=0 to R*4 : POKE V3+1,PEEK(54300) : NEXT I
```

Here is the schematic diagram for the filter. The stereo side is intended to be connected to the phono input.



Chapter 8 : Data Management

8.1 Introduction

The effective and efficient management and processing of data is one of the most basic themes in computing. All programs do it to some degree, but programs designed specifically for storage and retrieval of large quantities of various data are among the most complex in programming. It is the same in BASIC, FORTRAN, Pascal or other languages--data management, and everything else that has to do with data, is a very important problem. One would therefore expect that computer manuals or programming books would provide detailed information about this topic. Unfortunately, these books discuss data management only very briefly, it at all.

In this chapter we want to give you some insight into data management on the Commodore 64. We will not merely present dry theory, but we will also present examples which will hopefully allow you to understand your Commodore 64 better and to use it more effectively. First, however, we must begin by clarifying a few fundamentals of data management.

FILE

The whole world talks about data processing and files--but what actually are files? The easiest way to clarify this term is to replace it with another, one which everyone is familiar with: CARD CATALOG. As you know, a card catalog, such as those found in libraries, consists of a number of filing cards. On these cards is information concerning a particular item or person. The cards must be organized according to a specific pattern. The most common method of organization is alphabetic sorting. Another possibility is sorting by the item number or some other datum. All of the cards together make up a card catalog. A file uses the same principle. A file consists of a number of data records which contain the individual pieces of information--just like a filing card.

The great advantage of a data file over a card file is the amazing flexibility of the data file. The time savings when searching and sorting are most important, but the space savings also plays a large role today. Microcomputers can now be equipped with millions of characters of data storage. Can you imagine how many filing cards would be required to store so much information?

DATA RECORD

As we mentioned before, a data record can be compared to a filing card in a card catalog. Within this data record are all the data that would be on the filing card, divided into one or more FIELDS.

FIELD

Here too we can use the example of the card catalog. If you can picture a data record as a filing card, then the fields are the individual lines of information on the card. The association between the three concepts can be thought of approximately as:

FILE -> DATA RECORD -> FIELD

When one wants certain information about a thing or a person, such as the name, the appropriate file must first be accessed, then the data record from this file, and finally the appropriate field from the data record. This can be represented graphically as follows:

FILE: ADDRESSFILE

	FIELD LAST	FIELD FIRST	FIELD STREET	FIELD CITY	FIELD ZIP CODE
Rec: 1	Jones	Tom	123 Main St.	Anytown, AZ	55555
Rec: 2	Smith	John	456 Park Pl.	Nowhere, CA	86521
Rec: 3	Green	Joe	789 Kings Ct.	CBM City, TX	68513

.

.

In this example one can clearly see the differences and relationships between file, data record, and field. These terms should be well understood before one begins writing data management programs. We will now move on to various access and storage methods, but the basis for this presentation will be the material we have discussed so far.

8.2 Cassette - Diskette

After this somewhat lengthy introduction we want to actually write data management programs. We should first examine the devices which are at our disposal for storage on the Commodore 64: the datasette and disk drive.

How are these two devices and their media different? How can they be used? In order to clarify these questions we will first make an excursion into the beginnings of data processing.

Not so many years ago, terms such as "floppy" or "magnetic platter" were unheard. But even then one could not do without some sort of storage medium, a device that could save and recall data. Punch cards were developed for this purpose. With these one had a simple and cost-effective means of saving and retrieving data. A serious disadvantage of the devices required for working with the punch cards, the card puncher and reader, soon became apparent. Both were purely mechanical devices and far too slow. Faster and more reliable storage devices have always been in demand, so something better had to be developed. The result was the magnetic tape, which we can compare to the present cassette, since the principle is much the same as that used by the much larger reel-to-reel tape drives used on mainframe computers.

The principle of the cassette as storage medium is really quite simple. The Commodore 64 has a specific device assignment for the cassette, device number 1. The command for writing is also 1. To open a file on the cassette recorder, the following command might be used:

```
OPEN 1, 1, 1, "CBM 64 FILE"
```

The first "1" is the file number for the Commodore 64. If you want to open several files on the 64, you must choose different file numbers for the printer, cassette, and disk drive. The file number must be an integer from 1 to 255. When the record and play buttons on the cassette recorder are pressed, the Commodore 64 will write a special leader on the tape. This leader contains only the file name for a data file but can also include the start address if a program is being saved. This so-called program or file header is saved twice, after which the tape is stopped. At this point, data (or the program) can be saved.

The following sequence offers an additional possibility for saving a file:

```
OPEN 1, 1, 2, "CBM 64 FILE"
```

When you use this command, the computer will write one additional piece of information to the tape after the file. This information, called the EOT (End Of Tape), when encountered in a subsequent read, tells the computer that the tape ends at that point.

Once saved, data will be read in again at some time. The corresponding command is:

```
OPEN 1, 1, 0, "CBM 64 FILE"
```

The Commodore 64 will search for a particular file until it finds it or until it encounters an EOT marker.

IMPORTANT: When writing, the 64 does NOT search for a file with the given name. It writes without regard for any existing data at the exact point on the tape that it finds itself. For this reason, it is best to store only one file per cassette in order to prevent unintentional destruction of other data or programs.

After a while, magnetic tapes no longer sufficed (we will discuss the reasons why later), so the storage techniques were refined. Magnetic platters were used. Here too the Commodore 64 has a similar method of storage available. Here one can connect one or more devices called disk drives. The corresponding medium, the diskette, can be compared to a phonograph record. On both media there are various "tracks," although all of the material on the diskette is magnetic and therefore the tracks are invisible. This allows the "record" to not only be read from but also written to. The syntax of the command for writing or reading a file using the disk looks like this:

```
OPEN 2, 8, 2, "0:CBM 64 FILE,S,W"
```

or

```
OPEN 2, 8, 2, "0:CBM 64 FILE,S,R"
```

The first "2" is again the internal file number, "8" is the usual device number for a disk drive (but it can also be 9-12), and the second "2" is the channel number. The most interesting part, however, is the name. Here we find first the number of the disk drive (0 or 1), then the filename, then an "S" for "sequential file" (more about this later), and finally either a "W" for write or an "R" for read.

The most important differences between disk and cassette storage consist of

-Cost

Based on the initial purchase price, the cassette recorder is by far the cheaper storage medium, even though the price of the disk drive has come down dramatically and one can purchase a VIC-1541 for under $250 Another cost factor is the price of the actual storage media. For example, in order to store the 170,000 characters which will fit onto a single disk in the Commodore VIC-1541, one would need four C-60 cassettes. Here the cassette recorder offers no price advantage.

-Access time

Here the advantages of the disk are shown most clearly. For instance, reading a 10K program from a cassette requires 200 seconds, but only 20 for the VIC-1541 disk drive. To read a file consisting of 50 addresses, each 100 characters long, requires 180 seconds with a cassette recorder, while the disk drive requires only 18 seconds.

-Access methods and ease of programming and operation

While the cassette recorder allows only programs and sequential data files to be stored, the disk offers many more possibilities through its ability to make use of relative files (random access) and direct access. In addition, the disk is much easier to use. One need only give the disk drive the name of program to be saved, and the drive will find free space on the disk and save the program there.

It should now be clear that the cassette recorder is a low-cost device for the beginner or light user. Anyone who wants to use his computer for commercial purposes will require a disk drive.

Next, we will take a look at the under-lying technical principles of data storage on the cassette and then turn to the individual access methods and file forms.

The datasette, or HOW DO THE BITS GET ON THE TAPE?

Now that we have clarified the principles behind files (and you have hopefully understood them), we want to explain how the information is actually placed on the tape. The discussion will become a bit technical, but you may find the information useful nonetheless.

The method Commodore uses for representing the information (bits) on the cassette tape is called PPM or Pulse Position Modulation. This means that the Commodore 64, just like its other Commodore brothers, writes the digital signals directly, and not in the form of tone frequencies, to the tape. These digital signals are transmitted in three different lengths: short (S), long (L), and medium (M). From these three lengths, three different combinations are formed, which have the following meanings:

LLMM = BYTE ; this combination precedes every byte
MMSS = 1
SSMM = 0

The letter "A" would be represented on the tape in the following form:

LLMM	MMSS	SSMM	SSMM	SSMM	SSMM	SSMM	MMSS	SSMM	MMSS
BYTE	1	0	0	0	0	0	1	0	1
BIT#	0	1	2	3	4	5	6	7	parity ODD

The format of the whole program or file on the tape looks like this:

Programs	Data files
========	==========

Program header	File header
Start & end addresses, name	Name
Program header (again)	File header (again)
Program (a block)	Data block
Program (again)	Data block (again)
End block	End block

The header is constructed as follows:

Programs	Data files
========	==========
Start address (xxxx)	Start address (0000)
End address (xxxx)	End address (0000)
Program name (16 characters)	Filename (16 chars.)
Padding chars. (for prg. name)	Padding chars. (for filename)

A block consists of: (programs and data files)

approximately 2 seconds of leader

9-byte count down ($89 $88 $87 $86 $85 $84 $83 $82 $81) for first block

($09 $08 $07 $06 $05 $04 $03 $02 $01) for repetition

data

checksum (EXOR checksum for all data)

end marker (LLSS SSSS SSSS SSSS SSSS)

approximately 0.16 seconds of trailer

As we mentioned before, the method of representation is the same for all of the Commodore computers. The problem with exchanging data between the Commodore 64 and VIC-20 lies only in the different clock frequencies used. The system clock of the VIC-20 runs faster than that of the CBM, while the system clock of the CBM runs faster than that of the Commodore 64. In practice, this means that VIC-20 and Commodore 64 programs can be run on the larger CBM's, but that a VIC-20 program cannot be directly loaded into a Commodore 64, and vice versa. If you want to exchange cassettes between these two computers, you must make a detour through a CBM. The procedure would be something like this:

You have a cassette which contains one or more VIC-20 programs or files which you want to transfer to your Commodore 64. In order to do so, you first take a ordinary datasette and connect it to a CBM or PET computer and load the first program (or file) as usual. Then take a new cassette and exchange it for the VIC-20 cassette in the recorder. Now save you program (or file) onto the tape with the usual commands. If you have more than one program or file, repeat this procedure until you are done.

After all of the programs have been transferred, or perhaps it would be better to say converted, you have a cassette which can be used on both the VIC-20 and the Commodore 64. Under certain circumstances you may have to make some changes to the program, such as adapting it to the 64's 40x25 screen size, changing some POKE's, etc.

8.3 The principle behind data management: Sequential files

We have occupied ourselves in detail with the "history" of storage media. Now we shall turn our attention to the two most recent storage media, the tape and disk drives. In this section, we will concentrate on the sequential method of data access.

Sequential means "one after the other." This is exactly the way we find individual data records in the file. It can be compared with the selection of a piece of music with the aid of a cassette recorder. You fast-forward or rewind the tape to the specific place at which the piece is recorded and then press the play button. When working with sequential files, either on tape or disk, there is one additional limitation. It is like having a tape recorder with a fast-forward and a rewind-to-start-of-tape button. If we want to hear the piece of music again, or make another pass through our data, we must go back to the beginning of the tape and fast-forward to the desired spot again.

It works much the same way with data storage on the tape. When you save some data, you must make note of the counter position so that you can find the same spot on the tape later when you wish to read the data back in. You can use the fast-forward and rewind buttons to aid in finding the data. In spite of this, it is somewhat problematical to search through a file for a specific data record. If you have a file full of addresses, and you search for the name SMITH, it may happen that there is more than one SMITH in the file. Often, you cannot always make note of the counter position (in addition, we want to do without such manual work, otherwise we might just as well use filing cards). We must find some other way. We rewind the tape to the beginning, open the file for reading and then go through record by record until we have read the correct SMITH. Naturally, this has certain time expenses: with 2000 records in a file one can have a nice cup of coffee or walk the dog while the file is being processed. But one can still use the cassette recorder for working with small amounts of data, especially since it is very cost-effective for such applications.

Those who own cassettes but who would like to process their data quickly and efficiently should make use of the following procedure: Form all of your files such that they will fit into the free memory of the Commodore 64. Before you change, erase, insert, or simply display any of the data in the file, load the whole thing into memory from the cassette. Now the data accesses are not dependent on the speed of the cassette, rather, you can make use of the processing speed of your computer. When you are done working with the file, save the entire file back to tape. This simple and effective procedure can also be used for larger files. For example, you can divide an extensive address file into groups of names, one tape for those whose last names start with A-C, another for D-F, and so on, so that the parts each fit into the 64's memory. With some skill and organization a tape recorder can be used to manage a large amount of data.

When the process of data storage is presented figuratively, one can easily see why only sequential files are possible on the cassette recorder. All data are saved one after the other and read back into the computer in the same way.

Sequential files are also available as a method of data storage on the disk drive. Sequential files can be found quickly and directly without searching since the drive maintains a directory of the disk's contents and where the files can be found. This allows you to escape the tedious searching necessary with the cassette recorder.

How do we handle a sequential file on the Commodore 64? First we must open the file. We need the file number, device number, channel number, and filename. Once we have opened a file, we can read or write in the file with one command, but never both at once.

Without doing something additional we cannot write to a file which already exists. If, for example, you open the file "CBM 64 FILE" with the command

```
OPEN 2, 8, 2, "0:CBM 64 FILE,S,W"
```

and a file with the same name already exists on the diskette, you will receive the error message FILE EXISTS. The command must therefore be modified by placing an at-sign ("@") in front of the drive number. The command is then worded

```
OPEN 2, 8, 2, "@0:CBM 64 FILE,S,W"
```

and will cause any existing file with the same name to be overwritten.

This is important because even with a disk, no data can be changed in an existing sequential data file. To change any data, the file must be read into memory in its entirety, and after making the changes it must be rewritten to the diskette. Those who wish to use sequential files with the disk instead of the direct access files available should use the procedure described for use with the cassette recorder.

Sequential files on the disk drive offer a substantially higher rate of access and data transfer speed as well as automated operation. The disk offers yet another advantage, namely the ability to APPEND to a sequential file. This ability to append is very useful because it means that you do not have to read all of the data into the Commodore 64 and then write it back again in order to simply add new data to an existing file. A simple change of the OPEN command allows you to append data to the end of a sequential file. The OPEN command looks like this:

```
OPEN 2, 8, 2, "0:CBM 64 FILE,S,A"
```

Now all data written to the file will be added to the end.

This append option is unfortunately unavailable on the cassette drive. You must read in all of the data, then write it back out again, and finally add the new data before closing the file. It should now be obvious why we said that data management is far more convenient with the disk than with the cassette.

Following, you will find a set of model programs for simple sequential data management on a cassette or disk drive. The individual programs can be easily modified for your own uses. First the cassette version.

1. Writing the data

```
 10 REM ***********************************
 20 REM WRITING FIRST AND LAST NAMES TO TAPE
 30 REM VERSION FOR DATASETTE / COMMODORE 64
 40 REM ***********************************
 50 PRINT CHR$(147) : REM ERASE SCREEN
 52 PRINT "OPENING FILE FOR WRITING"
 54 PRINT
 56 OPEN 1,1,1,"CBM 64 FILE"
 60 INPUT "LAST NAME  : ";LN$
 70 INPUT "FIRST NAME : ";N$
 80 PRINT
 90 PRINT "WRITING - LAST NAME  = ";LN$
100 PRINT "        - FIRST NAME = ";N$
110 PRINT
120 PRINT#1,LN$
130 PRINT#1,N$
140 PRINT
150 INPUT "MORE (Y/N) ";YN$
155 PRINT
160 IF YN$="Y" THEN 60
170 IF YN$="N" THEN 200
180 PRINT "INVALID INPUT!"
190 GOTO 140
200 CLOSE 1
210 END
```

This program will save a desired number of first and last names to tape. Note that this program can only be used with a cassette recorder. The next program is the "opposite" of the first. It reads the data into the 64 and displays it on the screen (or printer). Before running this program you must rewind the tape to the start of the file you created with the above program.

2. Reading the data

```
 10 REM ***************************************
 20 REM READING FIRST AND LAST NAMES FROM TAPE
 30 REM VERSION FOR DATASETTE  /  COMMODORE 64
 40 REM ***************************************
 50 PRINT CHR$(147) : REM ERASE SCREEN
 52 PRINT "OPENING FILE FOR READING"
 54 PRINT
 56 OPEN 1,1,0,"CBM 64 FILE"
 60 INPUT#1,LN$
 70 INPUT#1,N$
 80 IF ST AND 64 THEN 130 : REM END OF FILE?
 90 PRINT "READING - LAST NAME  = ";LN$
100 PRINT "         - FIRST NAME = ";N$
110 PRINT
120 GOTO 60
130 PRINT "END OF FILE - LAST NAME  = ";LN$
140 PRINT "             - FIRST NAME = ";N$
150 CLOSE 1
160 END
```

This program reads all of the data previously saved by the first program and then displays it on the screen. If you want to send the data to the printer instead of the screen, you must change a few lines:

```
 58 OPEN 4,4
 90 PRINT#4,"READING - LAST NAME  = ";LN$
100 PRINT#4,"        - FIRST NAME = ";N$
110 PRINT#4
130 PRINT#4,"END OF FILE - LAST NAME  = ";LN$
140 PRINT#4,"            - FIRST NAME = ";N$
155 CLOSE 4
```

There is yet another possibility, namely the addition of data. As we mentioned earlier, there is no simple way to append data to the end of a cassette-based sequential file as there is for such file on disk. The file must be read into memory in its entirety, the tape rewound, the file opened again for writing, and the previously read data written back to the tape. At the end you may add the new data. The same procedure would also allow you to change or erase individual data.

3. Adding data

```
 10 REM ************************************
 20 REM ADDING FIRST AND LAST NAMES TO TAPE
 30 REM VERSION FOR DATASETTE / COMMODORE 64
 40 REM ************************************
 50 PRINT CHR$(147) : REM CLEAR SCREEN
 52 PRINT "OPENING FILE FOR READING"
 54 PRINT
 56 OPEN 1,1,0,"CBM 64 FILE"
 60 DIM LN$(100),N$(100) : I=1 : REM 100 NAMES MAX.
 70 INPUT#1,LN$: LN$(I)=LN$
 80 INPUT#1,N$ : N$(I)=N$
 90 IF ST AND 64 THEN 130
100 IF I=100 THEN 130
110 I=I+1
120 GOTO 70
130 EN=I
135 PRINT
140 PRINT "PLEASE REWIND THE TAPE."
150 PRINT
160 INPUT "DONE (Y/N) ";YN$
170 IF YN$="N" THEN 130
180 IF YN$="Y" THEN 210
190 PRINT "INVALID INPUT!"
200 GOTO 150
210 PRINT "OPENING FILE FOR APPENDING"
220 PRINT
230 OPEN 1,1,1,"CBM 64 FILE"
240 FOR I=1 TO EN
250 PRINT#1,LN$(I)
260 PRINT#1,N$(I)
270 NEXT I
```

```
280 PRINT "ADD DATA:"
290 PRINT
300 INPUT "LAST NAME  : ";LN$
310 INPUT "FIRST NAME : ";N$
320 PRINT
330 PRINT "WRITING - LAST NAME  = ";LN$
340 PRINT "        - FIRST NAME = ";N$
350 PRINT
360 PRINT#1,LN$
370 PRINT#1,N$
380 PRINT
390 INPUT "MORE (Y/N) ";YN$
400 IF YN$="Y" THEN 300
410 IF YN$="N" THEN 440
420 PRINT "INVALID INPUT!"
430 GOTO 380
440 CLOSE 1
450 END
```

Now you have a small address manager. To be sure, it lacks the addresses yet, but anyone with a bit of experience in programming will be able to expand the program to include this.

We shall now turn to sequential file management on the disk drive. Here too we will offer the three examples which we gave for the datasette. This will allow you to compare and contrast the two.

1. Writing the data

```
10 REM ************************************
20 REM WRITING FIRST AND LAST NAMES TO DISK
30 REM VERSION FOR VIC-1541 / COMMODORE 64
40 REM ************************************
50 PRINT CHR$(147) : REM CLEAR SCREEN
52 PRINT "OPENING FILE FOR WRITING"
54 PRINT
56 OPEN 2,8,2,"CBM 64 FILE,S,W"
60 INPUT "LAST NAME  : ";LN$
70 INPUT "FIRST NAME : ";N$
80 PRINT
90 PRINT "WRITING - LAST NAME  = ";LN$
100 PRINT "        - FIRST NAME = ";N$
110 PRINT
120 PRINT#2,LN$
130 PRINT#2,N$
140 PRINT
150 INPUT "MORE (Y/N) ";YN$
155 PRINT
160 IF YN$="Y" THEN 60
170 IF YN$="N" THEN 200
180 PRINT "INVALID INPUT!"
190 GOTO 140
200 CLOSE 2
210 END
```

Exactly as the previously-described program for the datasette, this program writes any number of first and last names to the diskette in sequential form. Naturally, this works only until the diskette, or better, the file space, is full. It requires a large amount of data to fill up a diskette, but one should still take care in programming that no error or program crash will occur when the disk is full. So that you receive the full impression of the capacity of the disk, we want to show you a small example:

The VIC-1541 disk drive can store a total of 174,848 bytes (characters) on a diskette. We can use the following amounts for files:

Sequential files : 168,656 characters
Relative files : 167,132 characters

A maximum of 144 programs and files can be saved.

Let's assume that we have written a complete address manager. For the sake of example, assume our program expects the following data:

Field	Length
Number	3
First name	20
Last name	20
Street address	25
City	25
State	2
Zip code	5
Telephone number	14
Notes	50

Our data record is 164 characters long. To this we add the RETURN characters for the end of the fields (CHR$(13)). We must add one more character for each field. This yields a total of 173 characters. How many records can we store on one diskette?

The calculation we need here look like this:

MAX = BYTES FOR SEQUENTIAL FILES / LENGTH OF A RECORD

or in our example:

168,656 / 173 = 974.8901734

Since it will be a little difficult to make use of 974.8901734 data records, and a little space on the diskette never hurts, we could store up to about 960 records. This number should suffice for most applications. If, however, you need to store more records, you must either write the program in such a way so that it can make use of multiple data disks or use a larger disk drive. In our example, using a Commodore 8250 drive would increase the storage capacity by a factor of 6 per drive. This would mean that the 8250 could store more than 5500 addresses.

This possibility is available to the Commodore 64. All that you need is the disk drive itself and an IEEE-488 interface for the 64.

2. Reading the data

Now on to reading the data. The program is virtually identical to the cassette version. For the sake of completeness we will present this program again:

```
10 REM ****************************************
20 REM READING FIRST AND LAST NAMES FROM DISK
30 REM VERSION FOR VIC-1541  /  COMMODORE 64
40 REM ****************************************
50 PRINT CHR$(147) : REM CLEAR SCREEN
52 PRINT "OPENING FILE FOR READING"
54 PRINT
56 OPEN 2,8,2,"CBM 64 FILE,S,R"
60 INPUT#2,LN$
70 INPUT#2,N$
80 IF ST AND 64 THEN 130 : REM END OF FILE?
90 PRINT "READING - LAST NAME  = ";LN$
100 PRINT "        - FIRST NAME = ";N$
110 PRINT
120 GOTO 60
130 PRINT "END OF FILE - LAST NAME  = ";LN$
140 PRINT "            - FIRST NAME = ";N$
150 CLOSE 2
160 END
```

As you see, this program has no important differences from the cassette version. The only significant difference in the ways in which the cassette and disk work with sequential files is the disk drive's ability to append to the end of a sequential file without having to read in and rewrite the old data.

3. Adding data

```
 10 REM *************************************
 20 REM ADDING FIRST AND LAST NAMES TO A FILE
 30 REM VERSION FOR VIC-1541  /  COMMODORE 64
 40 REM *************************************
 50 PRINT CHR$(147) : REM CLEAR THE SCREEN
 52 PRINT "OPENING FILE FOR APPENDING"
 54 PRINT
 56 OPEN 2,8,2,"CBM 64 FILE,S,A"
 60 INPUT "NAME      : ";LN$
 70 INPUT "          : ";N$
 80 PRINT
 90 PRINT "WRITING - LAST NAME  = ";LN$
100 PRINT "        - FIRST NAME = ";N$
110 PRINT
120 PRINT#2,LN$
130 PRINT#2,N$
140 PRINT
150 INPUT "MORE (Y/N) ";YN$
155 PRINT
160 IF YN$="Y" THEN 60
170 IF YN$="N" THEN 200
180 PRINT "INVALID INPUT!"
190 GOTO 140
200 CLOSE 2
210 END
```

As you have noticed, this program bears a strong resemblance to the program for writing the data--with one exception: The OPEN command was changed from

```
OPEN 2,8,2,"CBM 64 FILE,S,W"
```

to

```
OPEN 2,8,2,"CBM 64 FILE,S,A"
```

This ability of the disk allows relatively easy manipulation of sequential files.

At the close of this section, we would like to clarify the range of applications of sequential files. For data management where fast access and easy alteration of data is important, sequential files are used only under certain conditions. Sequential files files are used primarily when a file is to be created for a clearly defined purpose in a clearly defined form. An example is data exchange between computers. Sequential files can, in principle, be read by any computer provided the character sets (generally ASCII) are compatible. With relative files or direct access, the various disk operating systems use different means of managing the files and such an exchange is generally not possible. Another example is register files which are created once and then never changed, such as the book-keeping journal in accounting.

8.4 Copying files with one and two disk drives

As we have discussed, there are various ways of expanding, changing, or erasing sequential files. Sequential data management can be very simple--but it is looked upon as only a primitive method of saving and retrieving data.

In addition, it is sensible or even necessary to duplicate data or files so that after working with a file, a copy of data in its original condition is still available or so that should anything happen to one copy of the file, the other can still be used.

We will first discuss copying files. In our example, we assume that the file has been saved sequentially. There are several ways of copying this file. First, we could read the entire file into the 64's memory in order to copy the records into the new file. This method either requires a very large amount of memory (the diskette can contain up to 170,000 characters while the Commodore 64 can hold only about 30,000 along with BASIC and program) or is limited to small files. A "compromise" is possible where a certain number of records are read in by blocks and then written back to the diskette. We will keep to the simplest method however and read each record in and then write it back.

Our goal is to create a second file which, after the copying process, is identical to the original file. The only problem which we encounter here is that we must know how many fields each record has. In order to make the program easier to use, we also it to display which record it is working along with its fields.

```
50000 AD$="ORIGINAL FILE":
      REM  NAME OF THE FILE (CHANGE AS NEEDED)
50010 NF$="NEW FILE":
      REM  NAME OF NEW FILE       (        "         )
50011 NF$="@:"+NF$+",S,W":
      REM  WRITE NEW FILE
50020 INPUT "HOW MANY FIELDS PER RECORD ";NF$
50030 NF=VAL(NF$)
50040 DIM FT$(NF):
      REM  DIMENSION FIELD TITLES
50045 DIM DF$(NF):
      REM  DIMENSION DATA FILEDS
50050 FOR I=1 TO AF:
        REM  INPUT ALL FIELD TITLES
50060   INPUT FT$(I)
50070 NEXT I
50080 PRINT
50090 PRINT "COPYING IN PROGRESS."
50100 OPEN 1,8,2,AF$:
      REM  OPEN FILE FOR READING
50110 OPEN 2,8,3,NF$:
      REM  OPEN FILE FOR WRITING
50112 RN=1:
      REM  BEGIN WITH RECORD 1
50115 PRINT "READING RECORD NO.";RN:
      PRINT
50120 FOR I=1 TO NF:
        REM  READ ALL FIELDS
50130   INPUT#1,DF$(I)
50140   PRINT FT$(I);" : ";DF$(I)
50145   DL=ST:
        REM  DL = FILE STATUS
50150 NEXT I
50160 PRINT
50170 PRINT "WRITING RECORD NO.";NR:
      PRINT
50180 FOR I=1 TO NF:
        REM  WRITE ALL FIELDS
50190   PRINT#2,DF$(I)
50200 NEXT I
50210 PRINT
50220 IF DL AND 64
     THEN 51000:
        REM  END OF THE FILE?
50230 DR=DR+1:
      REM  NEXT RECORD
50240 GOTO 50115
51000 PRINT "ALL RECORDS COPIED."
51010 PRINT
51020 CLOSE 2:
      REM  CLOSE THE FILES
51030 CLOSE 1
51040 PRINT "END."
51050 END
```

With this program you can easily copy your own sequential files, so long as you know the construction (the number of fields per data record) of the file.

This routine has a problem, however. When you want to copy a very large file, you will very soon reach the limits of the disk drive's capacity. You can see that a 100,000 character file (about 100KB) cannot be copied so easily with this program since you cannot create a destination file which has both the same construction and size of the original.

In order to copy large files, we must either work with two disk drives or two different diskettes. The easiest and surest way is to make the duplicate using two disk drives. One of the drives must be defined as device 8 and the other as device 9. This can be done in software, although DIP switches inside the drive can be changed to make the device number more permanent.

Once you have defined one drive as device 9, you can alter the previous program so that even large files can be copied.

```
50110 OPEN 2,9,3,ND$: REM OPEN FILE FOR WRITING
```

Data will now be read from drive 8, displayed on the screen, and written back to drive 9.

Many programmers use record 0 or 1 of their file or a second file to store information about the construction of the file. It would, for example, be very useful if you would save the number of fields and number of records as the first information in the file. This makes it unnecessary to input this information manually when you want to copy the file. In addition, you always know how many records must still be copied. If you have saved these two values in the file, the program must naturally be changed somewhat.

```
50012 OPEN 1,8,2,AD$: REM OPEN ORIGINAL FILE FOR READING
50013 INPUT#1,NF$: REM NUMBER OF FIELDS
50014 INPUT#1,NR$: REM NUMBER OF RECORDS
50015 CLOSE 1
```

Delete line 50020

```
50035 NR=VAL(NR$)
50111 INPUT#1,NF$: INPUT#1,NR$: PRINT#2,NF$: PRINT#2,NR$
50112 FOR RN=1 TO NR
```

Delete lines 50145 and 50220

```
50230 NEXT RN
```

Such a parameterized file is considerably easier to work with.

Appending records can be done using the same principle. First the file must be copied with this program, then file 2, the new data file, is not closed but expanded with the usual PRINT# commands. Once the file is expanded as desired, you can copy it back to the original.

8.5 Faster access: Relative files

Other Commodore computers with BASIC 4.0 and Commodore 64's equipped with IEEE expanders with BASIC 4.0 or MASTER 64 have much easier methods of managing relative files than does an unaided Commodore 64. In a relative file, each record carries a number which, based on its position, is relative to the beginning of the file. This allows you to construct a data management program using one of two basic options:

1) You use the ordering criterium of the relative file, namely the given record number, as the access key for your record. Using this, you could set the account number in an account file equal to the record number. This makes possible a faster, more direct access to the desired account. The same applies for part numbers and other numeric keys which you may want to use.

2) You build a table which contains the keys indexed to the record numbers. If, for example, you have ordered your address file by names and want to search for an address with the name SMITH, you first search the table for the name SMITH and then using the record number associated with the name, access this record directly. This procedure is considerably faster and more elegant than reading through a sequential file until the name is found.

Unfortunately, users of serial-oriented Commodore 64's who have not added BASIC 4.0 capability to their machines cannot normally make use of these efficient relative files. The VIC-1541 disk drive's operating system is able to work with relative files, but the necessary commands are not available in the 64's Commodore BASIC 2.0. We would like to show you a way in which one can use relative files on the Commodore 64 in spite of this limitation.

The possibility exists to inform the disk drive using CHR$ commands which record is to be written or read. The whole procedure consists of two parts:

1) Opening the relative file with the usual OPEN command:

```
OPEN filenumber, deviceaddress, channelnumber,
                  "name,L,"+CHR$(length)
```

The first part of this OPEN command is the same as that for sequential files. After the declaration of the name comes an "L". This L stands for LENGTH--the disk drive now knows that it is supposed to open a relative file. Next comes a very important CHR$ command. This command tells the disk drive the length of the data records in our file. In our previous example of the address file, we would enter 173 here as the record length. The Commodore 64 and the disk operating system allow a maximum record length of 254 characters. If a record requires more than 254 characters, either another file must be opened and the record divided into two or more smaller records or you can write in the same file and make note of the fact that every second record is the second part of the "meta-record."

2. Positioning the record pointer:

```
PRINT# channelnumber,"P"+CHR$(channelnumber)+
                   CHR$(low)+CHR$(high)
```

The special part of this command begins after the declaration of the file number. The "P" means POSITION and tells the operating system that the following CHR$ commands are to set the record pointer through the input of LOW and HIGH (we will show you later how to calculate LOW and HIGH).

The command can be expanded even farther. If you add another CHR$ command to the end of the current string, this will designate the position within data record. This allows you to set the record pointer to a specific character.

There is one very important characteristic of relative files which must be noted:

A terminating character (CHR$(13)) must be written to the record after each FIELD is written. Without this separating character, the computer will not be able to distinguish between successive fields. For this reason we have always placed the PRINT# commands on different lines so that a carriage return, CHR$(13), is automatically saved between the records.

This will all be made clearer through an example. Therefore, we have included a completely functional inventory control program at the end of this section so that you can see the procedures discussed in the section actually used in a program. We believe that the trouble of typing this program in will be well worth it, since with only minor changes it can be used as an address manager, tape and record cataloguer, and more.

But first to the above-mentioned HIGH and LOW numbers. These HIGH and LOW numbers together give the actual data record number. The formula for calculating the record number is:

record number = HIGH * 256 + LOW

This allows us access to records with numbers greater than 255. To read the 78th record, for instance, we must first calculate HIGH and LOW:

HIGH = INT (record number / 256)

LOW = record number - HIGH * 256

or in a concrete example

```
HB=INT(78/256): LB=78-HB*256
```

which yields the values

HB = 0, LB = 78

This calculation is rather trivial for reading a record whose number is less than 256, but this example shows how all of the calculations can be made.

This result must now be used in the command to set the record pointer. To set the pointer to the 78th record, the command is

```
PRINT# channelnumber,"P"+CHR$(channelnumber)+CHR$(0)+CHR$(78)
```

In our inventory management program, you will find the following structure:

```
PRINT#15,"P"+CHR$(3)+CHR$(231)+CHR$(3)+CHR$(1)
```

This command will set the pointer to the first character within the 999th data record of the file.

Before the file can be used, it must first be prepared for relative operation. This is done by setting the record pointer to a record and then writing to this record with the character CHR$(255). This character tells the operating system that an existing data record lies at this point, in which nothing has yet been written. In our example, all of the 999 records are marked with this character.

Now we can write the record in this file, but no more than we have declared when we opened the file. If one tries to write a record which lies outside of the allowed range, the computer will respond with the error message RECORD NOT PRESENT, since this data record does not exist.

In our program, after you start it with RUN, you will be asked if the disk drive is connected. This message will appear until you press the Y key. After this, you will be asked if you want to use a new disk. "New" means only that the disk is unformatted or that it has not been initialized for the file. Be careful, though, because the disk will be formatted in any event, so don't use a disk which contains anything you might want to keep. When this is done, the main menu will appear. From this point, you can call up six possible functions.

When you want to construct a record, remember that the input may be no longer than the length given in the data lines (30-82). These lines are constructed such that the name is entered first and then the length of the field. To delete an existing record, go to the routine CHANGE and enter an @ as the first character in the DESCRIPTION field. This will mark the record as erased.

In any function, you can return to the main menu by entering END when asked to enter the part number.

When first entering a part, only the part number and description are entered. To enter an initial quantity, you must enter this quantity as a sales slip. At this time you will also have to opportunity to set cost and price of the item. This must also be done when receiving items. To update the inventory (when goods are sold), enter the quantity sold as a negative number when entering the sales slip. You do not have to re-enter the cost and price each time--just press RETURN when asked.

In addition, a printer and disk drive must always be connected when working with this program. If you do not have a printer, you must rewrite the program. Lines containing PRINT#4 commands must be changed since these are the lines which send the data to the printer.

If this program is to be used for multiple branches or by more than one person, it is recommended that a new disk be used for each branch or person.

We hope that this program will offer you some insight into data management, especially data management with relative files. It looks at first glance more difficult than it really is. With a little practice, you will be able to design similar programs of your own.

```
10      CLR
15      ME(1)=1.065:
        ME(2)=1.13:
        ME(3)=1.07:
        ME(4)=1.14
20      FOR I=1 TO 7:
          READ TD$(I),TD(I):
        NEXT I
30      DATA "1) PART NUMBER    :",3
32      DATA "2) DESCRIPTION    :",20
34      DATA "3) QUANTITY       :",3
36      DATA "4) COST/EACH      :",7
38      DATA "5) TOTAL COST     :",8
40      DATA "6) PRICE/EACH     :",7
42      DATA "7) TOTAL PRICE    :",8
50      FOR I=1 TO 3:
          READ TI$(I),TI(I):
        NEXT I
56      DATA "1) PART NUMBER    :",3
58      DATA "2) DESCRIPTION    :",20
60      DATA "                  :",1
70      FOR I=1 TO 4:
          READ TT$(I),TT(I):
        NEXT I
76      DATA "1) BRANCH NUMBER :",1
78      DATA "2) DATE           :",8
80      DATA "3) ACCOUNT NUMBER:",8
82      DATA "4) RECEIPT NUMBER:",8
100     PRINT CHR$(147)
110     PRINT "****************************************";
120     PRINT "*      DATA MANAGEMENT PROGRAM 1.0      *";
130     PRINT "****************************************";
140     PRINT :
        PRINT
150     PRINT "DISK DRIVE CONNECTED? ";
160     GET A$:
        IF A$=""
        THEN 160
170     IF A$<>"Y"
        THEN 160
180     PRINT A$
190     OPEN 15,8,15,"IO":
        CLOSE 15
200     PRINT "NEW DISKETTE? (Y/N)  ";
210     GET A$:
        IF A$=""
        THEN 210
220     IF A$<>"Y"
        THEN 300
222     PRINT A$
230     OPEN 15,8,15,"N:DATA DISKETTE,AH"
240     OPEN 1,8,3,"0:INVDAT,L,"+CHR$(64)
250     PRINT#15,"P"+CHR$(3)+CHR$(231)+CHR$(3)+CHR$(1)
260     PRINT#1,CHR$(255);
270     RM=INT(167132/64)
```

```
280    CLOSE 1:
       CLOSE 15
300    PRINT CHR$(147)
310    PRINT "****************************************";
320    PRINT "*     DATA MANAGEMENT PROGRAM 1.0      *";
330    PRINT "****************************************";
340    PRINT :
       PRINT
345    PRINT TAB(15);"MAIN MENU":
       PRINT :
       PRINT
350    PRINT "     1) ENTER PART":
       PRINT
355    PRINT "     2) CHANGE PART":
       PRINT
360    PRINT "     3) ENTER SALES SLIP":
       PRINT
365    PRINT "     4) PRINT PARTS LIST":
       PRINT
370    PRINT "     5) PRINT EVALUATION":
       PRINT
375    PRINT "     6) EXIT PROGRAM":
       PRINT :
       PRINT
380    PRINT "YOUR SELECTION (1-6) : ";
390    GET A$:
       IF A$=""
       THEN 390
400    A=VAL(A$):
       IF A<1 OR A>6
       THEN 390
410    PRINT A$
420    FOR I=1 TO 1000:
       NEXT
430    ON A
       GOTO 1000,2000,3000,4000,5000,6000
1000   OPEN 15,8,15:
       OPEN 8,8,8,"0:INVDAT"
1002   GOSUB 12000
1005   PRINT CHR$(147)
1010   PRINT "***************************************"
1020   PRINT "*     DATA MANAGEMENT PROGRAM 1.0     *"
1030   PRINT "***************************************"
1040   PRINT :
       PRINT
1050   PRINT TAB(7);"INPUT PART":
       PRINT :
       PRINT
1060   FOR I=1 TO 2
1065     TE$(I)=""
1070     PRINT TI$(I);
1080     INPUT TE$(I)
1090   NEXT
1092   IF TE$(1)="END"
       THEN 1200
```

```
1093  FOR I=1 TO 3
1095    IF LEN(TE$(I))>TI(I)
        THEN 1065
1100  NEXT
1102  FOR I=4 TO 8:
        TE$(I)="":
      NEXT
1110  RN=VAL(TE$(1))
1120  IF RN<1 OR RN>999
      THEN 1005
1130  GOSUB 10000
1140  GOSUB 10070
1150  GOTO 1005
1200  CLOSE 8:
      CLOSE 15
1220  GOTO 300
2000  OPEN 15,8,15:
      OPEN 8,8,8,"0:INVDAT"
2002  GOSUB 12000
2005  PRINT CHR$(147)
2010  PRINT "****************************************";
2020  PRINT "*      DATA MANAGEMENT PROGRAM 1.0      *";
2030  PRINT "****************************************";
2040  PRINT :
      PRINT
2050  PRINT TAB(8);"CHANGE PART":
      PRINT :
      PRINT
2055  TE$(1)=""
2060  PRINT TI$(1);
2070  INPUT TE$(1)
2080  PRINT
2090  IF TE$(1)="END"
      THEN 2400
2100  IF LEN(TE$(1))>TI(1)
      THEN 2055
2110  RN=VAL(TE$(1))
2120  IF RN<1 OR RN>999
      THEN 2005
2130  GOSUB 10000
2140  GOSUB 10030
2142  IF VAL(TE$(1))<>RN
      THEN 2005
2150  PRINT CHR$(147)
2160  PRINT "****************************************";
2170  PRINT "*      DATA MANAGEMENT PROGRAM 1.0      *";
2180  PRINT "****************************************";
2190  PRINT :
      PRINT
2200  PRINT TAB(8)"CHANGE PART":
      PRINT :
      PRINT
2210  FOR I=1 TO 2
2220    PRINT TI$(I);"? ";
2230    PRINT TE$(I)
2240    PRINT CHR$(145);
```

```
2250    PRINT TI$(I);
2260    INPUT TE$(I)
2270    PRINT
2280    IF TE$(1)="END"
        THEN 2400
2290    IF LEN(TE$(I))>TI(I)
        THEN 2250
2300   NEXT
2310   RN=VAL(TE$(1))
2320   IF RN<1 OR RN>999
       THEN 2005
2330   GOSUB 10000
2340   GOSUB 10070
2400   CLOSE 4:
       CLOSE 8:
       CLOSE 15
2430   GOTO 300
2530   GOTO 3005
3000   OPEN 15,8,15:
       OPEN 8,8,8,"0:INVDAT"
3001   OPEN 4,4:
       DV=1
3002   GOSUB 12000
3005   PRINT CHR$(147)
3010   PRINT "****************************************";
3020   PRINT "*      DATA MANAGEMENT PROGRAM 1.0      *";
3030   PRINT "****************************************";
3040   PRINT :
       PRINT
3050   PRINT TAB(7)"ENTER SALES SLIP":
       PRINT :
       PRINT
3060   TE$(1)=""
3070   PRINT TI$(1);
3080   INPUT TE$(1)
3090   PRINT
3100   IF TE$(1)="END"
       THEN 3700
3110   IF LEN(TE$(1))>TI(1)
       THEN 3060
3120   RN=VAL(TE$(1))
3130   IF RN<1 OR RN>999
       THEN 3005
3132   IF DW=1 AND RN>799
       THEN 3005
3134   IF DW=2 AND RN<800
       THEN 3000
3140   GOSUB 10000
3150   GOSUB 10030
3152   IF VAL(TE$(1))<>RN
       THEN 3005
3154   IF LEFT$(TE$(2),1)="@"
       THEN 3005
3160   PRINT CHR$(147)
3170   PRINT "****************************************";
3180   PRINT "*      DATA MANAGEMENT PROGRAM 1.0      *";
```

```
3190  PRINT "****************************************";
3200  PRINT :
      PRINT
3210  PRINT TAB(7)"ENTER SALES SLIP":
      PRINT :
      PRINT
3212  FOR I=1 TO 5
3214    TH$(I)=TE$(I+3)
3216  NEXT
3220  FOR I=1 TO 2
3230    PRINT TD$(I)"? "TE$(I)
3235    TX$(I)=TE$(I)
3240    PRINT
3250  NEXT
3255  TX$(3)=TE$(3)
3260  PRINT TD$(3);
3270  INPUT TX$(4)
3275  TE$(4)=TX$(4)
3280  PRINT
3285  IF VAL(TE$(4))<-999 OR VAL(TE$(4))>999
      THEN 3260
3287  IF LEN(TE$(4))>TD(3)
      THEN 3260
3290  PRINT TD$(4);
3295  TE$(5)=""
3300  INPUT TX$(5)
3305  TE$(5)=TX$(5)
3310  PRINT
3315  IF LEN(TE$(5))>TD(4)
      THEN 3290
3320  PRINT TD$(6);
3325  TE$(7)=""
3330  INPUT TX$(7)
3335  TE$(7)=TX$(7)
3340  PRINT
3345  IF LEN(TE$(7))>TD(6)
      THEN 3320
3346  TH=VAL(TE$(4))
3347  TH=TH+VAL(TH$(1))
3348  TE$(4)=STR$(TH)
3350  TH=VAL(TE$(5))*VAL(TE$(4))
3351  TX$(6)=STR$(TH)
3355  TE$(6)=STR$(TH)
3360  TH=VAL(TE$(7))*VAL(TE$(4))
3361  TX$(8)=STR$(TH)
3365  TE$(8)=STR$(TH)
3370  TH=VAL(TE$(5))
3371  IF VAL(TE$(4))<1
      THEN TH=-TH
3375  TE$(5)=STR$(TH)
3380  TH=VAL(TE$(7))
3381  IF VAL(TE$(4))<1
      THEN TH=-TH
3385  TE$(7)=STR$(TH)
3460  RN=VAL(TE$(1))
3470  GOSUB 10000
```

```
3480  GOSUB 10070
3485  FOR I=1 TO 8:
        TE$(I)=TX$(I):
        TX$(I)="":
      NEXT
3490  IF DW=0 AND VAL(TE$(1))<800
      THEN DW=1:
        GOSUB 5360:
        GOTO 3510
3500  IF DW=0 AND VAL(TE$(1))>799
      THEN DW=2:
        GOSUB 7005:
        GOTO 3520
3510  IF DW=1
      THEN GOSUB 5520:
        GOTO 3530
3520  IF DW=2
      THEN GOSUB 7120
3530  GOTO 3005
3700  IF DW=1
      THEN GOSUB 5590:
        GOTO 3800
3710  IF DW=2
      THEN GOSUB 7190
3800  DW=0:
      DV=0
3999  CLOSE 4:
      CLOSE 8:
      CLOSE 15:
      GOTO 300
4000  OPEN 15,8,15:
      OPEN 8,8,8,"0:INVDAT"
4002  GOSUB 12000
4005  PRINT CHR$(147)
4010  PRINT "****************************************";
4020  PRINT "*      DATA MANAGEMENT PROGRAM 1.0      *";
4030  PRINT "****************************************";
4040  PRINT :
      PRINT
4050  PRINT TAB(8)"PRINT PARTS LIST":
      PRINT :
      PRINT
4060  FOR I=1 TO 2
4070    TE$(I)=""
4080    PRINT TT$(I);
4090    INPUT TE$(I)
4100    PRINT
4110    IF TE$(1)="END"
        THEN 4999
4120    IF LEN(TE$(I))>TT(I)
        THEN 4070
4130  NEXT
4135  TE$(3)="":
      TE$(4)=""
4140  IF DV=1
      THEN RETURN
```

```
4200  PRINT CHR$(147)
4210  PRINT "***************************************";
4220  PRINT "*      DATA MANAGEMENT PROGRAM 1.0      *";
4230  PRINT "***************************************";
4240  PRINT :
      PRINT
4250  PRINT TAB(8)"PRINT PARTS LIST":
      PRINT :
      PRINT
4260  PRINT "IS THE PRINTER TURNED ON? (Y/N)  ";
4270  GET A$:
      IF A$=""
      THEN 4270
4280  IF A$<>"Y"
      THEN 4270
4290  PRINT A$
4300  OPEN 4,4
4310  PRINT#4,"BRANCH NO. ";TE$(1);
4330  PRINT#4,CHR$(16);"20";"DATE ";TE$(2);
4340  PRINT#4,CHR$(16);"40";"ACCOUNT NO.  ";TE$(3);
4345  PRINT#4,CHR$(16);"60";"RECEIPT NO.  ";TE$(4)
4350  PRINT#4
4360  PRINT#4,"PART NO.";
4375  PRINT#4,CHR$(16);"15";"QUANTITY";
4385  PRINT#4,CHR$(16);"25";"DESCRIPTION"
4390  PRINT#4,"-----------";
4405  PRINT#4,CHR$(16);"15";"---------";
4415  PRINT#4,CHR$(16);"25";"-----------"
4420  FOR RN=1 TO 999
4430    GOSUB 10000
4440    GOSUB 10030
4442    IF VAL(TE$(1))<>RN
        THEN 4480
4444    IF LEFT$(TE$(2),1)="@"
        THEN 4480
4450    PRINT#4,TE$(1);
4460    PRINT#4,CHR$(16);"15";TE$(4);
4470    PRINT#4,CHR$(16);"25";TE$(2)
4480  NEXT
4490  FOR I=1 TO 3
4492    PRINT#4
4494  NEXT
4999  CLOSE 4:
      CLOSE 8:
      CLOSE 15:
      GOTO 300
5000  OPEN 15,8,15:
      OPEN 8,8,8,"0:INVDAT"
5002  GOSUB 12000
5005  PRINT CHR$(147)
5010  PRINT "***************************************";
5020  PRINT "*      DATA MANAGEMENT PROGRAM 1.0      *";
5030  PRINT "***************************************";
5040  PRINT :
      PRINT
5050  PRINT TAB(9)"PRINT THE EVALUATION":
```

```
        PRINT :
        PRINT
5060    FOR I=1 TO 4
5070      TE$(I)=""
5080      PRINT TT$(I);
5090      INPUT TE$(I)
5100      PRINT
5110      IF TE$(1)="END"
          THEN 5999
5120      IF LEN(TE$(I))>TT(I)
          THEN 5070
5125      IF DV=0 AND I=2
          THEN I=4
5130    NEXT
5140    IF DV=1
        THEN RETURN

5200    PRINT CHR$(147)
5210    PRINT "****************************************";
5220    PRINT "*      DATA MANAGEMENT PROGRAM 1.0       *";
5230    PRINT "****************************************";
5240    PRINT :
        PRINT
5250    PRINT TAB(9)"PRINT EVALUATION":
        PRINT :
        PRINT
5252    SA=0:
        SE=0:
        SG=0:
        SF=0:
        SH=0
5255    FOR I=1 TO 4:
          M1(I)=0:
          M2(I)=0:
        NEXT
5260    PRINT "IS THE PRINTER TURNED ON? (Y/N)  ";
5270    GET A$:
        IF A$=""
        THEN 5270
5280    IF A$<>"Y"
        THEN 5270
5290    PRINT A$
5300    OPEN 4,4
5310    PRINT#4,"BRANCH NO.  ";TE$(1);
5330    PRINT#4,CHR$(16);"20";"DATE  ";TE$(2);
5335    IF DV=0
        THEN PRINT#4:
          GOTO 5350
5340    PRINT#4,CHR$(16);"40";"ACCOUNT NO.  ";TE$(3);
5345    PRINT#4,CHR$(16);"60";"RECEIPT NO.  ";TE$(4)
5350    PRINT#4
5355    IF DV=1
        THEN RETURN

5360    PRINT#4,"PART NO.";
5370    PRINT#4,CHR$(16);"10";"QUA.";
```

```
5375  PRINT#4,CHR$(16);"15";"DESCRIPTION";
5380  PRINT#4,CHR$(16);"40";"COST/EA";
5390  PRINT#4,CHR$(16);"50";"TOT COST";
5400  PRINT#4,CHR$(16);"60";"PRICE/EA";
5410  PRINT#4,CHR$(16);"70";"TOT PRIC"
5420  PRINT#4,"--------";
5425  PRINT#4,CHR$(16);"10";"----";
5430  PRINT#4,CHR$(16);"15";"------------";
5440  PRINT#4,CHR$(16);"40";"--------";
5450  PRINT#4,CHR$(16);"50";"---------";
5460  PRINT#4,CHR$(16);"60";"---------";
5470  PRINT#4,CHR$(16);"70";"---------"
5475  IF DV=1
      THEN RETURN
5480  FOR RN=1 TO 799
        GOSUB 10000
5490    GOSUB 10030
5500    IF VAL(TE$(1))<>RN
5510    THEN 5580
5515    IF LEFT$(TE$(2),1)="@"
        THEN 5580
5520    PRINT#4,TE$(1);
5525    PRINT#4,CHR$(16);"10";TE$(4);
5530    PRINT#4,CHR$(16);"15";TE$(2);
5540    PRINT#4,CHR$(16);"40";TE$(5);
5550    PRINT#4,CHR$(16);"50";TE$(6);
5560    PRINT#4,CHR$(16);"60";TE$(7);
5570    PRINT#4,CHR$(16);"70";TE$(8)
5571    SA=SA+VAL(TE$(4))
5572    SE=SE+VAL(TE$(5))
5573    SG=SG+VAL(TE$(6))
5574    SF=SF+VAL(TE$(7))
5575    SH=SH+VAL(TE$(8))
5576    IF VAL(TE$(3))<1 OR VAL(TE$(3))>4
        THEN 5579
5579    IF DV=1
        THEN RETURN
5580  NEXT
5590  PRINT#4,"--------";
5600  PRINT#4,CHR$(16);"10";"----";
5605  PRINT#4,CHR$(16);"15";"------------";
5610  PRINT#4,CHR$(16);"40";"---------";
5620  PRINT#4,CHR$(16);"50";"---------";
5630  PRINT#4,CHR$(16);"60";"---------";
5640  PRINT#4,CHR$(16);"70";"---------"
5650  PRINT#4,"TOTALS:";
5660  PRINT#4,CHR$(16);"09";SA;
5670  PRINT#4,CHR$(16);"39";SE;
5680  PRINT#4,CHR$(16);"49";SG;
5690  PRINT#4,CHR$(16);"59";SF;
5700  PRINT#4,CHR$(16);"69";SH
5775  FOR I=1 TO 3:
        PRINT#4:
      NEXT
5777  RETURN
5780  GOSUB 7000
5999  CLOSE 4:
```

```
        CLOSE 8:
        CLOSE 15:
        GOTO 300
6000    CLOSE 4:
        CLOSE 8:
        CLOSE 15
6030    PRINT CHR$(147)
6040    END
7000    S1=0:
        S2=0
7005    PRINT#4,"PART NO.";
7010    PRINT#4,CHR$(16);"15";"DESC";
7020    PRINT#4,CHR$(16);"40";"EXPENDITURES";
7030    PRINT#4,CHR$(16);"60";"RECEIPTS"
7040    PRINT#4,"--------";
7050    PRINT#4,CHR$(16);"15";"---";
7060    PRINT#4,CHR$(16);"40";"---------";
7070    PRINT#4,CHR$(16);"60";"---------"
7075    IF DV=1
        THEN RETURN
7080    FOR RN=800 TO 999
7090      GOSUB 10000
7100      GOSUB 10030
7110      IF VAL(TE$(1))<>RN
          THEN 7180
7120      PRINT#4,TE$(1);
7130      PRINT#4,CHR$(16);"15";TE$(2);
7140      PRINT#4,CHR$(16);"40";TE$(5);
7150      PRINT#4,CHR$(16);"60";TE$(7)
7160      S1=S1+VAL(TE$(5))
7170      S2=S2+VAL(TE$(7))
7175      IF DV=1
          THEN RETURN
7180    NEXT
7190    PRINT#4,"--------";
7200    PRINT#4,CHR$(16);"15";"---";
7210    PRINT#4,CHR$(16);"40";"---------";
7220    PRINT#4,CHR$(16);"60";"---------"
7230    PRINT#4,"TOTALS:";
7240    PRINT#4,CHR$(16);"39";S1;
7250    PRINT#4,CHR$(16);"59";S2
7260    FOR I=1 TO 3:
          PRINT#4:
        NEXT
7380    FOR I=1 TO 3:
          PRINT#4:
        NEXT
7999    RETURN
8000    PRINT CHR$(147)
8010    PRINT "****************************************";
8020    PRINT "*      DATA MANAGEMENT PROGRAM 1.0      *";
8030    PRINT "****************************************";
8040    PRINT :
        PRINT
8050    PRINT TAB(7);"ENTER SALES SLIP":
        PRINT :
        PRINT
```

```
8060  SA=0:
      SE=0:
      SG=0:
      SF=0:
      SH=0:
      S1=0:
      S2=0
8070  FOR I=1 TO 4
8072    M1(I)=0
8074    M2(I)=0
8076  NEXT
8080  PRINT "IS THE PRINTER TURNED ON? (Y/N) ";
8090  GET A$:
      IF A$=""
      THEN 8090
8100  IF A$<>"Y"
      THEN 8090
8110  PRINT A$
8120  OPEN 4,4
8130  RETURN
8500  PRINT CHR$(147)
8510  PRINT "****************************************";
8520  PRINT "*     DATA MANAGEMENT PROGRAM 1.0      *";
8530  PRINT "****************************************";
8540  PRINT :
      PRINT
8550  PRINT TAB(7);"ENTER SALES SLIP":
      PRINT :
      PRINT
8560  GOSUB 5060
8570  GOSUB 5310
8580  RETURN
10000 HB=INT(RN/256):
      LB=RN-HB*256
10010 PRINT#15,"P"+CHR$(8)+CHR$(LB)+CHR$(HB)+CHR$(1)
10015 GOSUB 12000
10020 RETURN
10030 INPUT#8,TE$(1),TE$(2),TE$(3),TE$(4),TE$(5),TE$(6),
      TE$(7),TE$(8)
10060 RETURN

10070 TE$=TE$(1)+CHR$(13)+TE$(2)+CHR$(13)+TE$(3)+CHR$(13
      )+TE$(4)+CHR$(13)
10072 TE$=TE$+TE$(5)+CHR$(13)+TE$(6)+CHR$(13)+TE$(7)+CHR
      $(13)+TE$(8)
10080 PRINT#8,TE$
10110 RETURN
12000 INPUT#15,X,X$,Y$.Z$
12010 IF X<>0
      THEN 12030
12020 RETURN

12030 PRINT X;X$;Y$;Z$:
      CLOSE 8:
      CLOSE 15
12040 FOR I=1 TO 6000:
      NEXT
12060 GOTO 100
```

8.6 Another method: Direct access

This method of accessing data on the diskette is unfortunately often ignored or overlooked. It is quite complicated but it has some very interesting aspects. What does direct access allow us to do?

1) Accessing files - random files

This method has something to do with sequential file management, but without the disadvantages, and also has something in common with relative files.

2) Accessing individual tracks on the disk

This method of access offers you possibilities which you had probably not thought of before, and whose purpose you may not yet see. We will discuss it in greater detail later.

1. Random files

In contrast to the sequential and relative files, a single block in a random file is 256 bytes long, and a total of 664 such blocks can be stored on a diskette. You can also store shorter records, such as 4 64-byte records in a block. The task of correctly accessing the exact location within the block falls now to the programmer. To use a random file, you must first open a sequential file in order make note of the tracks in which you have stored the data. You will need a total of three files:

1) Sequential file for the pointer
2) Command file
3) Data file for direct storage

```
 10 OPEN 4,8,4,"CBM 64 FILE,S,W": REM SEQUENT. FILE
 20 OPEN 15,8,15: REM COMMAND CHANNEL
 30 OPEN 5,8,5,"#": REM DATA FILE
 40 TE$="ABACUS SOFTWARE"
 50 PRINT#5,TE$;",";1: REM TEXT, RECORD #
 60 T=1: S=1: REM TRACK=1, SECTOR=1
 70 PRINT#15,"B-A:";0,T,S: REM DRIVE, TRACK, SECTOR
 80 INPUT#15,ER,NA$,TR,BL: REM READ ERROR
 90 IF ER=65 THEN T=TR: S=BL: GOTO 70
100 PRINT#15,"B-W:";5,0,T,S: REM WRITE RECORD
110 PRINT#4,T;",";S
120 CLOSE 5
130 CLOSE 15
140 CLOSE 4
150 END
```

What does this program do? First it opens the three required files, then defines some text which will later be written to the disk. This text is first written to the data buffer, after which the operating system searches for the next free block on the diskette. The search begins at track 1, sector 1, the start of the diskette (line 70). "B-A:" means Block-Allocate and attempts to allocate the block defined by the drive, track, and sector numbers. If this block is not free (it is being used by some other file, or perhaps another part of the current one) the operating system will search until it finds a free block. In order to see if the sought-after block is free or not, we must read the command channel. If the error code has the value 65, we know that the block was not free. Once the computer finds a free block, it then writes the data stored in the data buffer in the proper block on the diskette. After this, it writes the address of the block to the sequential file so that the record can be found again later. At the same time, the operating system also makes note of this block so that it will not be overwritten by other files. The files are closed and the program ends.

Equally interesting is the retrieval of the data:

```
 10 OPEN 4,8,4,"CBM 64 FILE"
 20 OPEN 15,8,15
 30 OPEN 5,8,5,"#"
 40 INPUT#4,T,S: REM READ THE ADDRESSES
 50 PRINT#15,"B-R:";5,0,T,S
 60 INPUT#5,TE$,RE
 70 PRINT#15,"B-F:";0,T,S
 80 CLOSE 5
 90 CLOSE 4
100 PRINT#15,"S:CBM 64 FILE"
110 CLOSE 15
```

After the file is opened, the address (track and sector) of the block in which the data is saved is read in. The block itself is read, and the block is freed once again with the Block-Free command in line 70. This is to be done only when the block is to be deleted. Finally, the sequential and data files are closed, the sequential file is scratched, and the command channel is closed.

2. Direct disk access

This access makes it possible, as the name suggests, to directly access any desired tracks and sectors on the disk, that is, to read from and write to the disk without opening any files. This allows you to read the directory, for example, without using the LOAD"$",8 command and thereby destroying any program in memory. Or you could change a program on the diskette without having to load it; even destroyed programs can, under certain circumstances, be repaired.

This method of access is also quite dangerous, so we would like to warn all those against it who do not possess a good working knowledge of the construction of the diskette, the directory, and the BAM. Entire files and even the whole disk can be destroyed very easily with this command. To find out more about these commands, we refer you to the VIC-1541 user's guide or to the book The Anatomy of the 1541 Disk Drive. If you want to experiment with these commands, be sure to do it on a disk which does not contain any data or programs you might want to keep.

Here is a list of the commands which can be used to directly access the blocks on the diskette:

Name	Use
Block-Read	"B-R:";channel;drive;track;block
Block-Write	"B-W:";channel;drive;track;block
Block-Allocate	"B-A:";drive;track;block
Block-Free	"B-F:";drive;track;block
Buffer-Pointer	"B-P:";channel;position

These are the most important commands for direct access. Their common trait is that they all access the disk controller directly, offering possibilities not available otherwise. Many of the more useful possibilities can be found in The Anatomy of 1541 Disk Drive.

8.6 Rescuing an improperly closed file

Admittedly it does not happen often, but when it does, it is very annoying and results in a loss of work.

What is "it"?

By "it" we mean something like the following:

With much effort you have organized your record collection and would like to store the titles on a disk so that you can find them quickly. The usual method simply involves saving the record titles and artists' or composers' names in one or more sequential files. You are now in the course of entering the desired data via the keyboard and are almost done (you have already entered 500 titles) when your spouse trip over the power cord. "Doesn't matter," you think, "The data's safe on the diskette."

You return to your program, change the OPEN command for your sequential file to APPEND (A instead of W) and try to continue, but the red light on the disk begins to flash, indicating an error! Puzzled, and a bit worried, you read the error channel and find the message "WRITE FILE OPEN". When you list the directory, you find an "*" in front of the type designation. This means that the file is still open for writing since no CLOSE followed the write accesses. The same thing happens when you remove a disk from the drive without first closing all of the write files.

The usual methods offer you no chance of recovering your data. Too bad about the record collection.

Since this happened to us often enough, we have developed a small program which makes it possible to make the destroyed files at least readable again. Once again, we have provided a description of the program operations and the variables used.

Variables:

E	Position of the filename within the directory sector
S	Sector number for the direct commands
T	Track number for the direct commands
TY	File type (derived from T$)
X	Index variable for isolating the file name
A$	Interim variable for constructing S$
F$	Filename
S$	Complete sector
T$	File type
X$	16-character expanded filename read from directory, later the actual filename
X1$	Duplicate of X$

Program operation:

70 Open a data channel for the direct access

80 Open the command channel

100 Input filename

110 Assignment of track and sector numbers. For the VIC-1541 and the CBM 4040, the directory begins on track 18, sector 1. For the CBM 8050 disk drive, it is stored on track 39. If you are using this drive, this line must be changed accordingly.

120 In this line, the disk sector specified by T and S is read from the diskette (drive 0) into the internal buffer on the disk drive.

150 The buffer contents are transferred to A$.

160 A sector can contain up to eight directory entries.

These are first searched for the desired filename before the next sector is read in.

170 Here the filename is isolated from the entry and placed in X$.

200-210 The end criterium for the actual length of the filename is CHR$(160) (shifted space). Here the filename is removed and placed back into X$.

220-230 If the filename is found, execution branches to line 300, otherwise the other entries in the sector are searched.

240-260 At the beginning of each sector stands the track and sector addresses of the following block, or if there is none (end of the directory), the track number is zero.

300-310 The file type (the byte from which the file type for the screen is generated) is isolated and placed in T$ while the numeric value is placed in T.

320 T=0 marks an empty directory entry.

360 The bit which is set here is the cause of the whole problem. This bit is used to indicate if a file was opened for writing or not. The asterisk on the screen is derived from this bit.

370 The entire sector, including marker for a closed file, is reconstructed.

390-410 Now the buffer pointer in the drive is reset, the sector is placed into the buffer, and the buffer contents are written back to the disk.

420-490 These lines serve to remind you how to proceed with rescuing the file.

This program is quite simple to use:
Load the program, insert the disk containing the file you wish to rescue into the drive (it must be drive 0 for a double drive), run the program, and enter the name of the file.
A limitation:
This procedure does not work with relative files because they are stored differently on the disk. A relative file can only be reconstructed with a great deal of work.

Once you have rescued your file, you should read it record by record and rewrite the data to a new file. This is necessary because although the file has been recovered, the logical end of the file is no longer recognized.
At this point you should stop the procedure and be sure to close the new file and then erase the defective file.

We hope that you will find this program useful but also that you do not have to use it often.

```
10    PRINT CHR$(147);
20    PRINT CHR$(5);
70    OPEN 2,8,2,"#":
       REM  DIRECT ACCESS
80    OPEN 15,8,15:
       REM  COMMAND CHANNEL
90    PRINT :
       PRINT
100   INPUT "FILENAME ";F$:
       PRINT :
       PRINT
110   T=18:
       S=1:
       REM  1541 DIRECTORY ** T=39 FOR CBM 8050
120   PRINT#15,"U1 2 0"T;S:
       REM  READ
130   S$="":
       REM  VARIABLE FOR READ SECTOR
150   FOR I=1 TO 255:
         GET #2,A$:
         S$=S$+LEFT$(A$+CHR$(0),1):
       NEXT
160   FOR I=0 TO 7:
         REM  8 ENTRIES
170     X$=MID$(S$,I*32+6,16):
         X1$=X$
180     REM  ISOLATE FILENAME
190     X=1
200     IF MID$(X$,X,1)<>CHR$(160)
         THEN X=X+1:
           IF X<17
           THEN 200
210     X$=LEFT$(X$,X-1)
220     IF X$=F$
         THEN E=I:
           GOTO 300
230   NEXT I
240   T=ASC(S$):
       S=ASC(MID$(S$,2,1))
250   REM  READ NEXT SECTOR
260   IF T<>0
       THEN 120
270   REM  END
280   PRINT "FILE "F$" NOT ON THIS DISKETTE"
290   CLOSE 2:
       CLOSE 15:
       END
300   T$=MID$(S$,E*32+3)
310   TY=ASC(T$) AND 15
320   IF TY=0
       THEN NEXT I:
         GOTO 240
330   IF TY<>4
       THEN 340
```

```
335  PRINT "RELATIVE FILES CANNOT BE RESCUED"
337  GOTO 290
340  TY$="DELSEQPRGUSRREL"
350  PRINT "FILE "X1$" "MID$(TY$,TY*3+1,3):
      PRINT
360  T$=CHR$(ASC(T$) OR 128)
370  S$=LEFT$(S$,E*32+2)+T$+MID$(S$,E*32+4)
380  REM  * ERASE AND REWRITE
390  PRINT#15,"B-P 2 0"T;S
400  PRINT#2,S$;
410  PRINT#15,"U2 2 0"T;S
420  CLOSE 2:
      CLOSE 15
425  PRINT "FILE DATA CAN NOW BE READ."
430  PRINT "AFTER COPYING THE VALID DATA,"
440  PRINT "THE FOLLOWING COMMANDS SHOULD"
450  PRINT "BE GIVEN:":
      PRINT
460  PRINT "OPEN 15,8,15"
470  PRINT CHR$(17)"PRINT#15,"CHR$(34)"S:"F$CHR$(34)
480  PRINT CHR$(17)"PRINT#15,"CHR$(34)"V0"CHR$(34)
490  PRINT CHR$(17)"CLOSE 15"
500  END
```

Chapter 9 : POKE's and other useful routines

9.1 Using the cassette buffer as program storage

If one wants to use a small machine language program in conjunction with BASIC, the question always arises concerning where such programs should be placed in memory. A section of memory must be chosen which will not be overwritten by BASIC programs or variables. From this viewpoint there are two possibilities.

The first possibility is that a section of memory can be chosen which BASIC does not use at all, and the second is that the start or end of the BASIC program storage area can be changed. Three areas are unused by BASIC. The first is the cassette buffer. It lies from address 828 to 1019 ($033C to $03FB). This area is used by a program only when data is saved to or read from the cassette recorder. It works very well for machine language programs up to 192 bytes long. If sprite 13, 14, or 15 is used, the cassette buffer will be used to store these. Another small area is from address 704 to 767 ($02C0 to $02FF) which is used for sprite 11 (64 bytes). A large 4K-byte area above the BASIC interpreter is located fro 49152 to 53247 ($C000 to $CFFF), which should suffice for even the longest machine language programs.

If a few memory locations are needed, there are 16 bytes "behind" the screen memory which can be used. The 64 has 1K = 1024 bytes of memory for the screen, but only 40*25 = 1000 are used for the video RAM. 24 bytes are then left over, 8 of which are used as pointers for the sprites. Sixteen bytes remain which you can use for your own purposes. These are located from 2024 to 2039 (hexadecimal $07E9 to $07F8).

If these areas are not enough or you need more data storage area, the BASIC program storage area can be decreased and the extra space used by machine language programs. You can lower the end of the BASIC program area (the usual method) or raise the start. Let's take a closer look at how this is done.

The BASIC interpreter has two pointers which point to the start and end of the BASIC storage. The start-of-BASIC pointer is located at 43/44 ($2B/$2C), the end at 55/56 ($37/$38). These values can be read with

```
PRINT PEEK(43)+256*PEEK(44)
PRINT PEEK(55)+256*PEEK(56)
```

The values are normally 2049 and 40960. To make room for a 1000 byte machine language program, we can lower the end of BASIC by 1000, leaving it at 39960. We can set the new value with POKE statements.

```
HB = INT (39960/256) : LB = 39960 - HB*256
POKE 55,LB : POKE 56,HB : CLR
```

The CLR command is necessary to ensure that you do not get false variable values. To move the start to 3049, the following commands are necessary:

```
HB = INT (3049/256) : LB = 3049 - HB*256
POKE 43, LB : POKE 44,HB : POKE 3049-1,0 : NEW
```

Here the NEW command is necessary to properly reset the other BASIC pointers.

9.2 Sorting strings

One task which every programmer encounters sooner or later is the sorting of data. These could be names, addresses, or rows of numbers. There are various known algorithms used for sorting, but all of them are time consuming when large amounts of data have to be sorted. The simplest procedures are also generally the slowest. If one needs a faster sort method, one must formulate the algorithm not in BASIC but in machine language. For such tasks, solutions in machine language are about 100 times faster than a comparable BASIC routine. The following program is designed to sort strings. In order to keep it short, the following conditions must be kept in mind:

1. The field to be sorted must be the first dimensioned with a DIM statement.
2. An empty string must follow the last array element to be sorted.

Point 2 has the advantage that even a partially filled array can be sorted without all of the empty strings being placed at the start of the array after sorting.

With these arrangements, the program is so short that we can store it in the cassette buffer. It is called simply with SYS 828. The program checks to make sure that the array is a one-dimensional string array. If this is not the case, the machine language program is immediately ended.

```
033C                  ORG 828
033C A0 00            LDY #0
033E B1 2F            LDA ($2F),Y         ;FIRST LETTER
0340 30 0D            BMI L1
0342 C8               INY
0343 B1 2F            LDA ($2F),Y         :SECOND LETTE
R
0345 10 08            BPL L1
0347 A0 04            LDY #4
0349 B1 2F            LDA ($2F),Y         ;DIMENSION
034B C9 01            CMP #1
034D F0 01            BEQ L2
034F 60          L1   RTS
0350 18          L2   CLC
0351 A5 2F            LDA $2F             ;ARRAY START
0353 69 07            ADC #7              ;PLUS 7
0355 85 6E            STA $6E
0357 A5 30            LDA $30
0359 69 00            ADC #0
035B 85 6F            STA $6F
035D A0 00       L3   LDY #0
035F B1 6E            LDA ($6E),Y
0361 F0 EC            BEQ L1              ;LENGTH ZERO,
 DONE
0363 85 22            STA $22
0365 C8          L4   INY
0366 B1 6E            LDA ($6E),Y
0368 99 22 00         BYT $99,$22,$00
036B             ;LDA $22,0 ;POINTER TO STRING
036B C0 02            CPY #2
036D D0 F6            BNE L4
036F A5 6E            LDA $6E
0371 85 71            STA $71
0373 A5 6F            LDA $6F
0375 85 72            STA $72
0377 18          L5   CLC
0378 A5 71            LDA $71
037A 69 03            ADC #3              ;ADD THREE
037C 85 71            STA $71
037E 90 02            BCC L6
0380 E6 72            INC $72
0382 A0 00       L6   LDY #0
0384 B1 71            LDA ($71),Y
0386 F0 3D            BEQ L13
0388 85 4D            STA $4D             ;COMPARE LENG
TH
038A C5 22            CMP $22             ;WITH FIRST L
ENGTH
038C 90 02            BCC L7
038E A5 22            LDA $22
0390 85 55       L7   STA $55             ;COMPARE LENG
TH
0392 C8          L8   INY
0393 B1 71            LDA ($71),Y
```

```
0050 0395 99 4D 00          BYT $99,$4D,$00
0051 0398              ; STA $4D,Y
0052 0398 C0 02             CPY #2
0053 039A D0 F6             BNE L8
0054 039C A0 00             LDY #0
0055 039E B1 23        L9   LDA ($23),Y          ;STRING COMPA
RE
0056 03A0 D1 4E             CMP ($4E),Y
0057 03A2 F0 04             BEQ L10              ;EQUAL, THEN
CONTINUE
0058 03A4 B0 0B             BCS L11              ;GREATER THAN
 EXCHANGE
0059 03A6 90 CF             BCC L5               ;SMALLER THEN
 NEXT STRING
0060 03A8 C8           L10  INY
0061 03A9 C4 55             CPY $55              ;ALL CHARACTE
RS EQUAL?
0062 03AB D0 F1             BNE L9
0063 03AD C4 22             CPY $22              ;FIRST STRING
 LONGER
0064 03AF B0 C6             BCS L5               ;NO THEN OK
0065 03B1 A0 02        L11  LDY #2
0066 03B3 B1 6E        L12  LDA ($6E),Y          ;SWAP STRING
POINTERS
0067 03B5 AA                TAX
0068 03B6 B1 71             LDA ($71),Y
0069 03B8 91 6E             STA ($6E),Y
0070 03BA 99 22 00          BYT $99,$22,$00
0071 03BD              ; STA $22,Y
0072 03BD 8A                TXA
0073 03BE 91 71             STA ($71),Y
0074 03C0 88                DEY
0075 03C1 10 F0             BPL L12
0076 03C3 30 B2             BMI L5
0077 03C5 18           L13  CLC                  ;POINTER TO N
EXT STRING
0078 03C6 A5 6E             LDA $6E
0079 03C8 69 03             ADC #3
0080 03CA 85 6E             STA $6E
0081 03CC 90 8F             BCC L3
0082 03CE E6 6F             INC $6F
0083 03D0 D0 8B             BNE L3

ASSEMBLY COMPLETE.
```

```
100 FOR I = 828 TO 977
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 160,  0,177, 47, 48, 13,200,177, 47, 16,  8,160
130 DATA   4,177, 47,201,  1,240,  1, 96, 24,165, 47,105
140 DATA   7,133,110,165, 48,105,  0,133,111,160,  0,177
150 DATA 110,240,236,133, 34,200,177,110,153, 34,  0,192
160 DATA   2,208,246,165,110,133,113,165,111,133,114, 24
170 DATA 165,113,105,  3,133,113,144,  2,230,114,160,  0
180 DATA 177,113,240, 61,133, 77,197, 34,144,  2,165, 34
190 DATA 133, 85,200,177,113,153, 77,  0,192,  2,208,246
200 DATA 160,  0,177, 35,209, 78,240,  4,176, 11,144,207
210 DATA 200,196, 85,208,241,196, 34,176,198,160,  2,177
220 DATA 110,170,177,113,145,110,153, 34,  0,138,145,113
230 DATA 136, 16,240, 48,178, 24,165,110,105,  3,133,110
240 DATA 144,143,230,111,208,139
250 IF S <> 17663 THEN PRINT "ERROR IN DATA!!" : END
260 PRINT "OK"
```

We can demonstrate the speed of the machine language program with a small test program.

The program creates a given number of strings made up of a given maximum number of random letters, displays these on the screen, sorts them, and then prints them again, together with the time required for the sort.

```
100 INPUT "NUMBER, LENGTH";N,L
110 DIM A$(N) : N=N-1
120 FOR I=0 TO N
130 FOR J=1 TO RND(1)*1
140 A$(I) = A$(I)+CHR$(RND(1)*26+65)
150 NEXT : NEXT
160 FOR I=0 TO N : PRINT A$(I) : NEXT
170 T=TI : SYS 828 : T=TI-T
180 PRINT "SORT TIME =" T/60 "SECONDS"
190 FOR I=0 to N : PRINT A$(I) : NEXT
```

Run this program with various lengths and numbers of strings and make note of the sort times. 100 strings can be sorted in less than one second. A comparable BASIC program would require minutes.

If you use this program in your programs, remember that the last element in the array must be an empty string and that the array must be the first dimensioned.

9.3 Minimum and maximum of numeric fields

When performing calculations with dimensioned variables, one often needs to know the smallest or largest value in the field. This calculation can of course be performed by a small BASIC loop, but this takes relatively long for large fields. This is a good case for using machine language. The program uses the same algorithm as the corresponding BASIC variant.

```
100 DIM A(N)
 ...
200 GOSUB 1000
 ...
1000 MIN = A(0)
1010 FOR I=1 TO N
1020 IF A(I) < MIN THEN MIN = A(I)
1030 NEXT
1040 RETURN
```

A field A is dimensioned from 0 to N. By calling the subroutine at line 1000, the minimum is calculated and returned in the variable MIN. If the maximum is desired, one need only replace line 1020 with

```
1020 IF A(I) > MAX THEN MAX = A(I)
```

and line 1000 with

```
1000 MAX = A(0)
```

The machine language program has another advantage over its

BASIC counterpart in that it is not restricted to a single variable (our example above is limited to the variable A). The program will work with real numbers as well as integer arrays and resides at address $C800.

```
C800                                               ;MIN/MAX FUNC
TION
C800                 INTFLG EQU 14                 ;FLAG FOR INT
EGER VARIABLE
C800                 STORE  EQU $26
C800                 ARRTAB EQU $2F                ;POINTER TO A
RRAY TABLE
C800                 ARREND EQU $31                ;POINTER TO E
ND OF ARRAYS
C800                 VARNAM EQU $45                ;VARIABLE NAM
E
C800                 TEMP   EQU $5F
C800                 SETARR EQU $B196              ;POINTER TO F
IRST ARRAY ELEMENT
C800                 MEMFAC EQU $BBA2              ;GET CONSTANT
S IN FAC
C800                 CMPARE EQU $BC5B              ;COMPARE CONS
TANTS WITH FAC
C800                 ERROUT EQU $A445
C800                 INT    EQU $14                ;STORAGE FOR
INTEGER VARIABLE
C800                 INTFLT EQU $B391              ;INTEGER TO F
AC
C800                        ORG $C800
C800 A6 2F           MINMAX LDX ARRTAB
C802 A5 30                  LDA ARRTAB+1           ;POINTER TO S
TART OF ARRAY TABLE
C804 86 5F           L3     STX TEMP
C806 85 60                  STA TEMP+1             ;RUNNING POIN
TER
C808 C5 32                  CMP ARREND+1
C80A D0 04                  BNE L1
C80C E4 31                  CPX ARREND             ;END OF ARRAY
 TABLE?
C80E F0 1D                  BEQ NOTFND
C810 A0 00           L1     LDY #0
C812 B1 5F                  LDA (TEMP),Y           ;FIRST LETTER
 OF THE NAME
C814 C8                     INY
C815 C5 45                  CMP VARNAM             ;COMPARE WITH
 DESIRED NAME
C817 D0 06                  BNE L2
C819 A5 46                  LDA VARNAM+1
C81B D1 5F                  CMP (TEMP),Y           ;COMPARE SECO
ND CHARACTER
```

```
C81D F0 17              BEQ FOUND           ;FOUND?
C81F C8         L2      INY
C820 B1 5F              LDA (TEMP),Y
C822 18                 CLC
C823 65 5F              ADC TEMP            ;ADD OFFSET F
OR NEXT ARRAY
C825 AA                 TAX
C826 C8                 INY
C827 B1 5F              LDA (TEMP),Y
C829 65 60              ADC TEMP+1
C82B 90 D7              BCC L3
C82D A9 B8      NOTFND  LDA #<ERRMSG        ;POINTER TO E
RROR MESSAGE
C82F 85 22              STA $22
C831 A9 C8              LDA #>ERRMSG
C833 4C 45 A4           JMP ERROUT          ;OUTPUT ERROR
 MESSAGE
C836 C8         FOUND   INY
C837 18                 CLC
C838 B1 5F              LDA (TEMP),Y
C83A 65 5F              ADC TEMP
C83C 85 26              STA STORE
C83E C8                 INY
C83F B1 5F              LDA (TEMP),Y
C841 65 60              ADC TEMP+1
C843 85 27              STA STORE+1         ;POINTER TO E
ND OF THE ARRAY
C845 C8                 INY
C846 B1 5F              LDA (TEMP),Y        ;DIMENSION
C848 20 96 B1           JSR SETARR          ;POINTER TO F
IRST ARRAY ELEMENT
C84B 85 5F              STA TEMP
C84D 84 60              STY TEMP+1          ;SAVE POINTER
C84F 24 0E              BIT INTFLG          ;TEST TYPE
C851 30 24              BMI INTGER
C853 10 09              BPL LP1
C855 20 5B BC   L5      JSR CMPARE          ;COMPARE ARRA
Y ELEMENTS
C858 10 07              BPL LOOP
C85A A5 5F              LDA TEMP
C85C A4 60              LDY TEMP+1
C85E 20 A2 BB   LP1     JSR MEMFAC          ;SAVE ARRAY E
LEMENT AS MIN/MAX
C861 18         LOOP    CLC
C862 A5 5F              LDA TEMP
C864 69 05              ADC #5              ;POINTER TO N
EXT ELEMENT
C866 85 5F              STA TEMP
C868 90 02              BCC L4
C86A E6 60              INC TEMP+1
C86C A4 60      L4      LDY TEMP+1
```

```
0073 C86E C5 26          CMP STORE          ;END OF THE A
RRAY
0074 C870 D0 E3          BNE L5
0075 C872 C4 27          CPY STORE+1
0076 C874 D0 DF          BNE L5
0077 C876 60             RTS
0078 C877 A0 00   INTGER LDY #0             ;INTEGER ARRA
Y
0079 C879 B1 5F          LDA (TEMP),Y
0080 C87B AA             TAX
0081 C87C C8             INY
0082 C87D B1 5F          LDA (TEMP),Y
0083 C87F 85 15          STA INT+1          ;GET FIRST VA
LUE IN INT
0084 C881 86 14          STX INT
0085 C883 18      L12    CLC
0086 C884 A5 5F          LDA TEMP
0087 C886 69 02          ADC #2             ;POINTER TO N
EXT ELEMENT
0088 C888 85 5F          STA TEMP
0089 C88A 90 02          BCC L10
0090 C88C E6 60          INC TEMP+1
0091 C88E C5 26   L10    CMP STORE
0092 C890 D0 0D          BNE L11
0093 C892 A5 60          LDA TEMP+1
0094 C894 C5 27          CMP STORE+1        ;END REACHED?
0095 C896 D0 07          BNE L11
0096 C898 A5 14          LDA INT            ;GET INTEGER
VALUE
0097 C89A A4 15          LDY INT+1
0098 C89C 4C 91 B3       JMP INTFLT         ;CONVERT TO F
AC
0099 C89F A0 00   L11    LDY #0
0100 C8A1 B1 5F          LDA (TEMP),Y
0101 C8A3 C5 14          CMP INT            ;COMPARE HIGH
 BYTE
0102 C8A5 D0 07          BNE L14
0103 C8A7 C8             INY
0104 C8A8 B1 5F          LDA (TEMP),Y
0105 C8AA E5 15          SBC INT+1          ;COMPARE LOW
BYTE
0106 C8AC F0 D5          BEQ L12
0107 C8AE A9 01   L14    LDA #1             ;FLAG FOR GRE
ATER
0108 C8B0 90 02          BCC L13
0109 C8B2 A9 FF          LDA #$FF           ;FLAG FOR SMA
LLER
0110 C8B4 30 C1   L13    BMI INTGER
0111 C8B6 10 CB          BPL L12
0112 C8B8 41 52 52 ERRMSG ASC 'ARRAY NOT FOUN'
          41 59 20
          4E 4F 54
          20 46 4F
          55 4E
0113 C8C6 C4             BYT $C4
```

```
100 FOR I = 51200 TO 51398
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 166, 47,165, 48,134, 95,133, 96,197, 50,208,  4
130 DATA 228, 49,240, 29,160,  0,177, 95,200,197, 69,208
140 DATA   6,165, 70,209, 95,240, 23,200,177, 95, 24,101
150 DATA  95,170,200,177, 95,101, 96,144,215,169,184,133
160 DATA  34,169,200, 76, 69,164,200, 24,177, 95,101, 95
170 DATA 133, 38,200,177, 95,101, 96,133, 39,200,177, 95
180 DATA  32,150,177,133, 95,132, 96, 36, 14, 48, 36, 16
190 DATA   9, 32, 91,188, 16,  7,165, 95,164, 96, 32,162
200 DATA 187, 24,165, 95,105,  5,133, 95,144,  2,230, 96
210 DATA 164, 96,197, 38,208,227,196, 39,208,223, 96,160
220 DATA   0,177, 95,170,200,177, 95,133, 21,134, 20, 24
230 DATA 165, 95,105,  2,133, 95,144,  2,230, 96,197, 38
240 DATA 208, 13,165, 96,197, 39,208,  7,165, 20,164, 21
250 DATA  76,145,179,160,  0,177, 95,197, 20,208,  7,200
260 DATA 177, 95,229, 21,240,213,169,  1,144,  2,169,255
270 DATA  48,193, 16,203, 65, 82, 82, 65, 89, 32, 78, 79
280 DATA  84, 32, 70, 79, 85, 78,196
290 IF S <> 22908 THEN PRINT "ERROR IN DATA!!" : END
300 PRINT "OK"
```

The version printed here calculates the maximum of an array. If you want to calculate the minimum, you must reverse the branch logic after the comparisons. The contents of the following addresses must be changed:

```
C858    from $10 to $30
C8B4    from $30 to $10
C8B6    from $10 to $30
```

To use the function, you must first set the address for the USR function:

```
POKE 785,0 : POKE 786,200
```

Now you can call the function with PRINT USR(A) in which A is the name of the array. The USR function can be called as any other, for example X = USR(A%)*SIN(3).

The following small program will serve to demonstrate the function.

```
100 POKE 785,0 : POKE 786,200
110 INPUT "ARRAY SIZE ";N
120 DIM A(N)
130 FOR I=0 TO N
140 A(I) = RND (1)*1000
150 PRINT A(I)
160 NEXT
170 PRINT
180 PRINT USR(A)
```

The switch from MAX to MIN functions can be made by changing the three previously-mentioned values with POKE statements:

```
POKE 51288,48   (or back to 16)
POKE 51380,16   (or back to 48)
POKE 51382,48   (or back to 16)
```

9.4 DUMP command for variable output

The following machine language program is very useful for debugging BASIC programs. It prints out all of the BASIC variables together with their values. The program is stored in the cassette buffer and is called with SYS 828.

```
033C                ORG 828         ;CASSETTE BUF
FER
033C A5 2D          LDA $2D
033E A4 2E          LDY $2E         ;POINTER TO S
TART OF VARIABLES
0340 85 14     L0   STA $14         ;SAVE
0342 84 15          STY $15
0344 C4 30          CPY $30         ;COMPARE WITH
 END OF VARIABLES
0346 D0 02          BNE L1
0348 C5 2F          CMP $2F
034A B0 18     L1   BCS L3          ;TO END, THEN
 DONE
034C 69 02          ADC #2          ;POINTER TO V
ARIABLE VALUE
034E 90 01          BCC L2
0350 C8             INY
0351 85 22     L2   STA $22
0353 84 23          STY $23
0355 20 82 03       JSR L7          ;OUTPUT NAME
0358 20 B6 03       JSR L12         ;OUTPUT '='
035B 8A             TXA
035C 10 07          BPL L4
035E 20 BF 03       JSR L13         ;OUTPUT INTEG
ER VARIABLE
0361 4C 71 03       JMP L6          ;TO MAIN LOOP
0364 60        L3   RTS
0365 98        L4   TYA
0366 30 06          BMI L5
0368 20 CF 03       JSR L14         ;OUTPUT FLOAT
ING-POINT NUMBER
036B 4C 71 03       JMP L6
036E 20 D8 03  L5   JSR L16         ;OUTPUT STRIN
G VARIABLE
0371 A9 0D     L6   LDA #13         ;CARRIAGE RET
URN
0373 20 D2 FF       JSR $FFD2       ;OUTPUT
0376 A5 14          LDA $14
0378 A4 15          LDY $15
037A 18             CLC
```

```
0032 037B 69 07            ADC #7              ;ADD 7 FOR NE
XT VARIABLE
0033 037D 90 C1            BCC L0
0034 037F C8               INY
0035 0380 B0 BE            BCS L0              ;TO MAIN LOOP
0036 0382 A0 00      L7    LDY #0
0037 0384 B1 14            LDA ($14),Y         ;FIRST LETTER
 OF NAME
0038 0386 AA               TAX
0039 0387 29 7F            AND #$7F
0040 0389 20 D2 FF         JSR $FFD2           ;OUTPUT
0041 038C C8               INY
0042 038D B1 14            LDA ($14),Y         ;SECOND CHARA
CTER
0043 038F A8               TAY
0044 0390 29 7F            AND #$7F
0045 0392 F0 03            BEQ L8
0046 0394 20 D2 FF         JSR $FFD2           ;OUTPUT
0047 0397 8A         L8    TXA
0048 0398 10 11            BPL L10             ;TEST TYPE
0049 039A 98               TYA
0050 039B 30 0A            BMI L9
0051 039D A9 21            LDA #'!'            ;FUNCTION, OU
TPUT '!'
0052 039F 20 D2 FF         JSR $FFD2
0053 03A2 68               PLA
0054 03A3 68               PLA
0055 03A4 4C 71 03         JMP L6              ;JUMP BACK TO
 MAIN LOOP
0056 03A7 A9 25      L9    LDA #'%'            ;INTEGER VARI
ABLE
0057 03A9 D0 4E            BNE L19
0058 03AB 98         L10   TYA
0059 03AC 10 04            BPL L11
0060 03AE A9 24            LDA #'$'            ;STRING VARIA
BLE
0061 03B0 D0 47            BNE L19
0062 03B2 60         L11   RTS
0063 03B3 20 D2 FF         JSR $FFD2           ;OUTPUT CHARA
CTER
0064 03B6 A9 20      L12   LDA #$20
0065 03B8 20 D2 FF         JSR $FFD2           ;OUTPUT BLANK
0066 03BB A9 3D            LDA #'='
0067 03BD D0 3A            BNE L19             ;OUTPUT
0068 03BF A0 00      L13   LDY #0              ;INTEGER VARI
ABLE
0069 03C1 B1 22            LDA ($22),Y         ;LOW BYTE
0070 03C3 AA               TAX
0071 03C4 C8               INY
0072 03C5 B1 22            LDA ($22),Y         ;HIGH BYTE
0073 03C7 A8               TAY
0074 03C8 8A               TXA
0075 03C9 20 95 B3         JSR $B395           ;CONVERT TO F
LOATING POINT
```

```
03CC 4C D2 03          JMP L15          ;AND OUTPUT
03CF 20 A6 BB   L14    JSR $BBA6        ;GET FLOATING
-POINT VARIABLE
03D2 20 DD BD   L15    JSR $BDDD        ;CONVERT TO A
SCII STRING
03D5 4C 1E AB          JMP $AB1E        ;AND OUTPUT
03D8 20 F7 03   L16    JSR L18          ;OUTPUT STRIN
G, QUOTE
03DB A0 02             LDY #2
03DD B1 22             LDA ($22),Y      ;ADDRESS HIGH
03DF 85 25             STA $25
03E1 88                DEY
03E2 B1 22             LDA ($22),Y      ;ADDRESS LOW
03E4 85 24             STA $24
03E6 88                DEY
03E7 B1 22             LDA ($22),Y      ;LENGTH
03E9 85 26             STA $26
03EB F0 0A             BEQ L18
03ED B1 24      L17    LDA ($24),Y      ;OUTPUT CHARA
CTERS
03EF 20 D2 FF          JSR $FFD2        ;OF STRING
03F2 C8                INY
03F3 C4 26             CPY $26          ;STRING DONE?
03F5 D0 F6             BNE L17
03F7 A9 22      L18    LDA #$22         ;QUOTE
03F9 4C D2 FF   L19    JMP $FFD2        ;OUTPUT
```

```
100 FOR I = 828 TO 1019
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 165, 45,164, 46,133, 20,132, 21,196, 48,208,  2
130 DATA 197, 47,176, 24,105,  2,144,  1,200,133, 34,132
140 DATA  35, 32,130,  3, 32,182,  3,138, 16,  7, 32,191
150 DATA   3, 76,113,  3, 96,152, 48,  6, 32,207,  3, 76
160 DATA 113,  3, 32,216,  3,169, 13, 32,210,255,165, 20
170 DATA 164, 21, 24,105,  7,144,193,200,176,190,160,  0
180 DATA 177, 20,170, 41,127, 32,210,255,200,177, 20,168
190 DATA  41,127,240,  3, 32,210,255,138, 16, 17,152, 48
200 DATA  10,169, 33, 32,210,255,104,104, 76,113,  3,169
210 DATA  37,208, 78,152, 16,  4,169, 36,208, 71, 96, 32
220 DATA 210,255,169, 32, 32,210,255,169, 61,208, 58,160
230 DATA   0,177, 34,170,200,177, 34,168,138, 32,149,179
240 DATA  76,210,  3, 32,166,187, 32,221,189, 76, 30,171
250 DATA  32,247,  3,160,  2,177, 34,133, 37,136,177, 34
260 DATA 133, 36,136,177, 34,133, 38,240, 10,177, 36, 32
270 DATA 210,255,200,196, 38,208,246,169, 34, 76,210,255
280 IF S <> 20988 THEN PRINT "ERROR IN DATA!!" : END
290 PRINT "OK"
```

If you run the following program, you will receive the output shown below it.

```
100 A=5
110 DEF FNX (Y) = SIN(Y) * COS(Y)
120 C$ = "PROGRAM"
130 B% = -101
140 SYS 828
```

```
A = 5
X!
Y = 0
C$ ="PROGRAM"
B% =-101
```

You can also execute the DUMP function in the direct mode with SYS 828. If you stop a program, you can view the actual variable contents and then continue with the program using the CONT command. As you see in the above example, user-defined functions are indicated by a "!" after the function name.

9.5 Modified PEEK function

The following small machine language program provides an elegant way of using the additional RAM storage of the Commodore 64. At the same time, it also allows you to read the character generator data from BASIC. A few clarifications:

The memory areas from $A000 to $BFFF (40960 to 49151) and $E000 to $FFFF (57344 to 65535) are doubly allocated: First with 8K BASIC ROM and 8K kernal ROM, respectively, and then with 8K of RAM each. These 16K bytes of RAM cannot be used from BASIC without modification. POKE commands write directly to the RAM, but a read attempt with PEEK always reads from the ROM. Here we replace the PEEK function with our own USR function. The function must do the following: Before the value of a memory location is read, the memory configuration must be changed so that the RAM "beneath" the ROM is activated. Now the value can be read. Finally, the old configuration must be restored. In addition, we would like to be able to read the character generator which resides from location $D000 to $DFFF. The routine checks to see if the PEEK address lies between $D000 and $DFFF. If so, the memory configuration will be set such that the character generator can be read. The value is then read and the memory configuration returned to normal.

```
033C                                     ;USR - PEEK
033C             ADR    EQU $14          ;INTEGER ADDR
ESS
033C             FACADR EQU $B7F7
033C             YFAC   EQU $B3A2
033C                    ORG 828          ;CASSETTE BUF
FER
033C A5 14              LDA ADR
033E 48                 PHA              ;SAVE INTEGER
 ADDRESS
033F A5 15              LDA ADR+1
0341 48                 PHA
0342 20 F7 B7           JSR FACADR       ;CONVERT FAC
TO ADDRESS FORMAT
0345 A5 01              LDA 1
0347 48                 PHA              ;SAVE CONFIGU
RATION
0348 A5 15              LDA ADR+1
034A C9 D0              CMP #$D0         ;SMALLER THAT
 $D000
034C 90 07              BCC RAM
034E C9 E0              CMP #$E0         ;GREATER THAN
 $E000
0350 B0 03              BCS RAM
0352 A9 31              LDA #$31         ;READ FROM CH
ARACTER GENERATOR
0354 2C                 BYT $2C
0355 A9 34       RAM    LDA #$34         ;READ FROM RA
M
0357 78                 SEI
0358 85 01              STA 1            ;SET MEMORY C
ONFIGURATION
035A A0 00              LDY #0
035C B1 14              LDA (ADR),Y      ;READ BYTE
035E A8                 TAY
035F 68                 PLA
0360 85 01              STA 1            ;GET CONFIGUR
ATION
0362 58                 CLI
0363 68                 PLA
0364 85 15              STA ADR+1
0366 68                 PLA              ;GET ADDRESS
BACK
0367 85 14              STA ADR
0369 4C A2 B3           JMP YFAC         ;CONVERT Y TO
 FAC
```

The program is stored in the cassette buffer at address 828. Once you have entered or loaded the program, the start address of the program must be assigned to the USR vector. This is done with two POKEs:

```
POKE 785, 828 AND 255
POKE 786, 828 / 256
```

For those who do not have an assembler, we have again provided a loader program in BASIC, which also initializes the USR vector for you.

```
100 FOR I = 828 TO 875
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 165, 20, 72,165, 21, 72, 32,247,183,165,  1, 72
130 DATA 165, 21,201,208,144,  7,201,224,176,  3,169, 49
140 DATA  44,169, 52,120,133,  1,160,  0,177, 20,168,104
150 DATA 133,  1, 88,104,133, 21,104,133, 20, 76,162,179
160 IF S <> 5085 THEN PRINT "ERROR IN DATA!!" : END
170 POKE 785, 828 AND 255 : POKE 786, 828/256
180 PRINT "OK"
```

Now, if you want to read from the RAM or character generator, you simply replace the PEEK function with the USR function. To read the character matrix of a character, for example, you could use the following program:

```
100 CG=13*4096
110 A = (PEEK(53248+24) AND 2) * 1024
120 INPUT "CHARACTER CODE ";C
130 FOR I=0 TO 7
140 PRINT I, USR(CG+A+8*C+I) : NEXT
150 GOTO 110
```

Line 110 chooses between the upper or lower half of the character generator which selects between the upper case/graphics set or the upper/lower case set.

This new "PEEK" function gives you up to 16K of RAM which you can use to store data in BASIC or whatever else you like.

9.6 Multi-tasking on the Commodore 64

Multi-tasking is a term originally associated with mainframe computers and refers to the ability of a computer to execute several programs simultaneously. How does something like this work?

Even a mainframe can only do one thing at a time, so another trick is used:

If, for example, the computer is supposed to run five programs at once, it will start executing the first the program and after a certain length of time (a fraction of a second) will stop executing it, save the variables, and start executing the next program. This program too will be interrupted after a short time and the computer will continue executing the next one. Once all of the programs have been executed once, the variables from the first program are fetched and the execution of this program continues. The computer's time is divided up into "time slices" among the various programs. The term "time-sharing" is also used to describe this.

In a limited sense, this sort of thing is also possible on the Commodore 64. Two programs within the 64 run simultaneously: the BASIC interpreter and the so-called interrupt service routine which is called and executed 60 times per second. While your BASIC program is being executed, it is being interrupted 60 times a second in order to execute this interrupt routine. This routine takes care of such things as reading the keyboard.

Here we can attach our own routine and perform additional tasks of our own during the interrupt. One use of this might be to output text on the printer. At each interrupt a character could be fetched from a buffer and sent to the printer. The user could then continue with his

BASIC program as usual.

As an example of this procedure we have written a program which displays the time, including tenths of a second, on the screen, even while another program is running. The program uses the Commodore 64's real-time clock. The time is automatically and constantly displayed in the upper right-hand corner of the screen. The program is written in machine language but can also be entered using the BASIC loader program listed after the assembly language source code.

```
C800                                       ;TIME DISPLAY
C800                                       ;SYS AD,"HHMMSS",COLOR
C800                                       ;
C800                 FRMEVL EQU $AD9E      ;GET BASIC EXPRESSION
C800                 FRESTR EQU $B6A3
C800                 CHKCOM EQU $AEFD      ;CHECK FOR COMMA
C800                 CHRGOT EQU $79
C800                 GETBYT EQU $B79E      ;GET BYTE EXPRESSION
C800                 ILLQUA EQU $B248      ;'ILLEGAL QUANTITY'
C800                 ADR    EQU $22
C800                 COLOR  EQU $2A7       ;STORAGE FOR COLOR VALUE
C800                 VIDEO  EQU $288       ;HI BYTE VIDEO RAM
C800                 TEMP   EQU $FB
C800                 IRQ    EQU $314       ;IRQ VECTOR
C800                 PNT    EQU $FB
C800                 IRQVEC EQU $EA31      ;NORMAL IRQ VECTOR
C800                 CLR    EQU $D800      ;COLOR RAM
C800                 TENTHS EQU $DC08      ;REAL TIME CLOCK CIA 1
C800                 SECOND EQU TENTHS+1
C800                 MINUTE EQU SECOND+1
C800                 HOURS  EQU MINUTE+1
C800                 TRIGER EQU HOURS+3    ;50/60 HZ
C800                 SET    EQU TRIGER+1   ;SET TIME/ALARM
C800                        ORG $C800
C800 AD 0E DC               LDA TRIGER
C803 09 80                  ORA #$80       ;50 HZ MODE
```

```
C805 8D 0E DC          STA TRIGER
C808 AD 0F DC          LDA SET
C80B 29 7F             AND #$7F          ;SET TIME
C80D 8D 0F DC          STA SET
C810 20 79 00          JSR CHRGOT        ;ADDITIONAL CODE?
C813 F0 65             BEQ CHGIRQ        ;SWITCH CLOCK
C815 20 FD AE          JSR CHKCOM
C818 20 9E AD          JSR FRMEVL        ;GET STRING
C81B 20 A3 B6          JSR FRESTR        ;PARAMETER
C81E C9 06             CMP #6            ;6 CHARACTERS?
C820 D0 6B             BNE ILL           ;ILLEGAL QUANTITY
C822 A0 00             LDY #0
C824 B1 22             LDA (ADR),Y
C826 38                SEC
C827 E9 30             SBC #'0'          ;TO HEX
C829 C9 03             CMP #3
C82B B0 60             BCS ILL
C82D 0A                ASL
C82E 0A                ASL
C82F 0A                ASL
C830 0A                ASL
C831 85 FB             STA TEMP
C833 C8                INY
C834 B1 22             LDA (ADR),Y
C836 38                SEC
C837 E9 30             SBC #'0'
C839 C9 0A             CMP #10
C83B B0 50             BCS ILL
C83D 05 FB             ORA TEMP
C83F D0 04             BNE NOTNUL
C841 A9 92             LDA #$92          ;12 O'CLOCK PM
C843 D0 0F             BNE SETSTD
C845 C9 24     NOTNUL  CMP #$24
C847 B0 44             BCS ILL
C849 C9 13             CMP #$13
C84B 90 07             BCC SETSTD
C84D 38                SEC
C84E F8                SED
C84F E9 12             SBC #$12
C851 D8                CLD
C852 09 80             ORA #$80          ;SET PM
C854 8D 0B DC  SETSTD  STA HOURS
C857 20 FD C8          JSR GET59         ;GET MINUTES
C85A 8D 0A DC          STA MINUTE
C85D 20 FD C8          JSR GET59
C860 8D 09 DC          STA SECOND
C863 A9 00             LDA #0
C865 8D 08 DC          STA TENTHS        ;START CLOCK
C868 20 79 00          JSR CHRGOT
C86B F0 0D             BEQ CHGIRQ
C86D 20 FD AE          JSR CHKCOM
```

```
C870 20 9E B7          JSR GETBYT          ;COLOR
C873 E0 10             CPX #16
C875 B0 16             BCS ILL
C877 8E A7 02          STX COLOR           ;SAVE COLOR C
ODE
C87A 78         CHGIRQ SEI                 ;EXCHANGE IRQ
 VECTORS
C87B AD 14 03          LDA IRQ
C87E 49 A1             EOR #$A1            ;#<IRQVEC EOR
 TIMIRQ
C880 8D 14 03          STA IRQ
C883 AD 15 03          LDA IRQ+1
C886 49 22             EOR #$22            ;#>IRQVEC EOR
 TIMIRQ
C888 8D 15 03          STA IRQ+1
C88B 58                CLI
C88C 60                RTS
C88D 4C 48 B2   ILL    JMP ILLQUA
C890                                       ;DISPLAY ROUT
INE
C890 A5 FB      TIMIRQ LDA PNT
C892 48                PHA
C893 A5 FC             LDA PNT+1           ;SAVE POINTER
C895 48                PHA
C896 AD 88 02          LDA VIDEO           ;HIGH BYTE OF
 VIDEO RAM
C899 85 FC             STA PNT+1
C89B A9 00             LDA #0
C89D 85 FB             STA PNT             ;POINTER TO V
IDEO RAM
C89F A0 1E             LDY #30             ;30TH COLUMN
C8A1 AD 0B DC          LDA HOURS
C8A4 C9 12             CMP #$12
C8A6 F0 11             BEQ ZEROCK
C8A8 C9 80             CMP #$80
C8AA 90 0F             BCC STDOUT          ;AM
C8AC 29 7F             AND #$7F
C8AE C9 12             CMP #$12
C8B0 F0 09             BEQ STDOUT
C8B2 F8                SED
C8B3 18                CLC
C8B4 69 12             ADC #$12
C8B6 D8                CLD
C8B7 D0 02             BNE STDOUT
C8B9 A9 00      ZEROCK LDA #0
C8BB 20 DB C8   STDOUT JSR PRINT           ;DISPLAY HOUR
S
C8BE AD 0A DC          LDA MINUTE
C8C1 20 DB C8          JSR PRINT           ;DISPLAY MINU
TES
C8C4 AD 09 DC          LDA SECOND
C8C7 20 DB C8          JSR PRINT           ;DISPLAY SECO
NDS
C8CA AD 08 DC          LDA TENTHS
C8CD 09 30             ORA #'0'
```

```
0123 C8CF 20 F3 C8          JSR PRINT1          ;DISPLAY TENT
HS
0124 C8D2 68                PLA
0125 C8D3 85 FC             STA PNT+1
0126 C8D5 68                PLA                 ;GET POINTER
BACK
0127 C8D6 85 FB             STA PNT
0128 C8D8 4C 31 EA          JMP IRQVEC          ;TO OLD IRQ
0129 C8DB 48        PRINT   PHA                 ;DISPLAY
0130 C8DC 29 F0             AND #$F0
0131 C8DE 4A                LSR
0132 C8DF 4A                LSR
0133 C8E0 4A                LSR
0134 C8E1 4A                LSR
0135 C8E2 18                CLC
0136 C8E3 69 30             ADC #'0'
0137 C8E5 20 F3 C8          JSR PRINT1
0138 C8E8 68                PLA
0139 C8E9 29 0F             AND #$0F
0140 C8EB 18                CLC
0141 C8EC 69 30             ADC #'0'
0142 C8EE 20 F3 C8          JSR PRINT1
0143 C8F1 A9 3A             LDA #':'
0144 C8F3 91 FB     PRINT1  STA (PNT),Y         ;CHARACTER
0145 C8F5 AD A7 02          LDA COLOR
0146 C8F8 99 00 D8          STA CLR,Y           ;AND COLOR
0147 C8FB C8                INY
0148 C8FC 60                RTS
0149 C8FD C8        GET59   INY
0150 C8FE B1 22             LDA (ADR),Y
0151 C900 38                SEC
0152 C901 E9 30             SBC #'0'
0153 C903 C9 06             CMP #6
0154 C905 B0 86     ILL1    BCS ILL
0155 C907 0A                ASL
0156 C908 0A                ASL
0157 C909 0A                ASL
0158 C90A 0A                ASL
0159 C90B 85 FB             STA TEMP
0160 C90D C8                INY
0161 C90E B1 22             LDA (ADR),Y
0162 C910 38                SEC
0163 C911 E9 30             SBC #'0'
0164 C913 C9 0A             CMP #10
0165 C915 B0 EE             BCS ILL1
0166 C917 05 FB             ORA TEMP
0167 C919 60                RTS
```

```
100 FOR I = 51200 TO 51481
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 173, 14,220,  9,128,141, 14,220,173, 15,220, 41
130 DATA 127,141, 15,220, 32,121,  0,240,101, 32,253,174
140 DATA  32,158,173, 32,163,182,201,  6,208,107,160,  0
150 DATA 177, 34, 56,233, 48,201,  3,176, 96, 10, 10, 10
160 DATA  10,133,251,200,177, 34, 56,233, 48,201, 10,176
170 DATA  80,  5,251,208,  4,169,146,208, 15,201, 36,176
180 DATA  68,201, 19,144,  7, 56,248,233, 18,216,  9,128
190 DATA 141, 11,220, 32,253,200,141, 10,220, 32,253,200
200 DATA 141,  9,220,169,  0,141,  8,220, 32,121,  0,240
210 DATA  13, 32,253,174, 32,158,183,224, 16,176, 22,142
220 DATA 167,  2,120,173, 20,  3, 73,161,141, 20,  3,173
230 DATA  21,  3, 73, 34,141, 21,  3, 88, 96, 76, 72,178
240 DATA 165,251, 72,165,252, 72,173,136,  2,133,252,169
250 DATA   0,133,251,160, 30,173, 11,220,201, 18,240, 17
260 DATA 201,128,144, 15, 41,127,201, 18,240,  9,248, 24
270 DATA 105, 18,216,208,  2,169,  0, 32,219,200,173, 10
280 DATA 220, 32,219,200,173,  9,220, 32,219,200,173,  8
290 DATA 220,  9, 48, 32,243,200,104,133,252,104,133,251
300 DATA  76, 49,234, 72, 41,240, 74, 74, 74, 74, 24,105
310 DATA  48, 32,243,200,104, 41, 15, 24,105, 48, 32,243
320 DATA 200,169, 58,145,251,173,167,  2,153,  0,216,200
330 DATA  96,200,177, 34, 56,233, 48,201,  6,176,134, 10
340 DATA  10, 10, 10,133,251,200,177, 34, 56,233, 48,201
350 DATA  10,176,238,  5,251, 96
360 IF S <> 32970 THEN PRINT "ERROR IN DATA!!" : END
370 PRINT "OK"
```

Once you have loaded the program, the clock can be turned on by entering the following command:

```
SYS 51200,"HHMMSS",COLOR
```

where "HHMMSS" is the current time (Hours, Minutes, Seconds) and COLOR is the color code for the time display (from 0 to 15). To set the clock to 2:30 P.M. (since this is a 24-hour clock we must enter 14:30) and 15 seconds, with the time displayed in yellow, we would use the following command:

```
SYS 51200, "143015",7
```

The current time will now appear in the upper-right corner of the display with hours, minutes, seconds, and tenths of seconds. To turn the display off, enter

```
SYS 51200
```

To turn it back on again without resetting the time or color, simply type

```
SYS 51200
```

and the time will appear again.

In principle there are two methods for inserting the second "job" in multi-tasking:

The first option is to use the system interrupt routine which is called every sixtieth of a second. This method is used for our routine to display the time. This is done by changing the interrupt vector so that it points to our routine. Our routine then ends with a jump to the original interrupt routine so that the computer can complete its operations.

The second method gives the user routine its own interrupt. This could be done with the output to the printer, for example. The BUSY line of the printer could be used as the interrupt source. Each time the printer is ready to receive a character it initiates an interrupt. The interrupt routine sends a character to the printer then continue with the normal program. Once the printer has printed the character, it generates another interrupt, forcing the computer to send it another character. The user of the computer notices nothing of this.

You will need to know quite a bit about the operating system of the 64 to implement these routines, information which you will find in the book The Anatomy of the Commodore 64.

9.7 POKEs and zero page

As you have surely noticed, there are various addresses which are of use in programming in BASIC as well as in machine language. Here is a short list of some of the addresses (all of the pointers are stored in LSB, MSB order):

Address:	(possible) Application:
0000-0001	A specific area of memory can be switched on or off by POKEing to one or both of these locations.
0043-0044	These addresses point to the start of the user storage, the start of the BASIC program. PEEK(43)+256*PEEK(44) will show you this value. You can set the beginning higher by poking to these locations and use the lower memory area for the rest of the sprites.
0045-0046	In these addresses you find the start of the numeric variable table. This table usually lies directly behind the BASIC storage.
0047-0048	Contain the address of the start of the array storage. All fields (arrays) are placed in this area.
0049-0050	The contents of these addresses point to the end of the array storage.

0051-0052	In these locations is the pointer to the start of the BASIC string variables.
0055-0056	Pointer to the end of the BASIC RAM. By changing the contents of these addresses it is possible to protect a specific section of RAM (above the BASIC storage) against overwriting. This allows you to reserve this protected memory for a machine language program and still have the RAM from $C000 to $CFFF free for other purposes. For example: POKE 55,0 : POKE 56,64 sets the end of BASIC RAM to $4000.
0115-0138	The CHRGET routine resides at these addresses. This routine gets the characters from the individual BASIC lines. In order to write BASIC expansions, this routine must be altered.
0203	The code for the currently pressed key is stored in this address. If this address contains 64, it means that no key was pressed.

If you want to learn more about the "insides" of the Commodore 64, we recommend the Abacus book The Anatomy of the Commodore 64. There you will learn more about programming in machine language and the construction of the 64's RAM and ROM. We encourage you to experiment with the various addresses of the Commodore 64. There is much hidden in your computer--it only needs to be drawn forth.

SERIOUS 64 SOFTWARE

INDISPENSIBLE TOOLS FOR YOUR COMMODORE 64

PASCAL-64

This full compiler produces fast 6502 machine code. Supports major data Types: REAL, INTEGER, BOOLEAN, CHAR, multiple dimension arrays, RECORD, FILE, SET and pointer. Offers easy string handling, procedures for sequential and relative data management and ability to write INTERRUPT routines in Pascal! Extensions included for high resolution and sprite graphics. Link to ASSEM/MON machine language. **DISK $39.95**

DATAMAT-64

This powerful data base manager handles up to 2000 records per disk. You select the screen format using up to 50 fields per record. DATAMAT 64 can sort on multiple fields in any combination. Complete report writing capabilities to all COMMODORE or ASCII printers. **DISK $39.93**
Available November

TEXTOMAT-64

This complete word processor displays 80 columns using horizontal scrolling. In memory editing up to 24,000 characterrs plus chaining of longer documents. Complete text formatting, block operations, form letters, on-screen prompting.
Available November **DISK $39.95**

ASSEMBLER / MONITOR-64

This complete language development package features a macro assembler and extended monitor. The macro assembler offers freeform input, complete assembler listings with symbol table (label), conditional assembly.

The extended monitor has all the standard commands plus single step, quick trace breakpoint, bank switching and more. **DISK $39.95**

BASIC-64

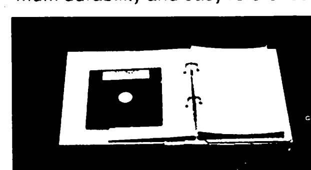
This is a full compiler that won't break your budget. Is compatible with Commodore 64 BASIC. Compiles to fast machine code. Protect your valuable source code by compiling with BASIC 64.
Available December **DISK $39.95**

ADA TRAINING COURSE

This package is an introduction to ADA, the official language ot the Department of Defense and the programming language of the future. Includes editor, syntax checker/compiler and 110 page step by step manual describint the language.
Available November **DISK $79.95**

OTHER NEW SOFTWARE COMING SOON!

All software products featured above have inside disk storage pockets, and heavy 3-ring-binder for maximum durability and easy reference.



DEALER INQUIRIES INVITED

AVAILABLE AT COMPUTER STORES, OR WRITE:

Abacus Software

P.O. BOX 7211 GRAND RAPIDS, MI 49510

Exclusive U.S. DATA BECKER Publishers

For postage & handling, add $4.00 (U.S. and Canada), add $6.00 for foreign. Make payment in U.S. dollars by check, money order of charge card. (Michigan Residents add 4% sales tax.)

FOR QUICK SERVICE PHONE (616) 241-5510

Commodore 64 is a reg. T.M. of Commodore Business Machines

TRICKS & TIPS FOR THE COMMODORE 64

This collection of easy-to-use programming techniques is "required" reading for every '64 owner. We present advanced graphics, easy data input, enhanced **BASIC** commands, **CP/M for the '64**, **POKES** and other routines and much more. We include dozens of ready-to-use hints, suggestions and programs.

ISBN 0-916439-03-8

YOU CAN COUNT ON
Abacus Software

P.O. BOX 7211 GRAND RAPIDS, MICH. 49510 PHONE 616-241-5510