CHAPTER 3

BASIC FUNCTIONS

The intrinsic functions provided by BASIC are presented in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y	Represent any numeric expressions
I and J	Represent integer expressions
X$ and Y$	Represent string expressions

If a floating point value is supplied where an integer is required, BASIC will round the fractional portion and use the resulting integer.

N O T E

Only integer and single precision results are returned by functions. Double precision functions are not supported.

3.1 ABS

Format: ABS(X)

Action: Returns the absolute value of the expression X.

Example:
```
PRINT ABS(7*(-5))
 35
Ok
```

3.2 ASC

Format: ASC(X$)

Action: Returns a numerical value that is the ASCII code of the first character of the string X$. (See Appendix E for ASCII codes.) If X$ is null, an "Illegal function call" error is returned.

Example:
```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

See the CHR$ function for ASCII-to-string conversion.

3.3. ATN

Format: ATN(X)

Action: Returns the arctangent of X in radians. Result is in the range -pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example:
```
10 INPUT X
20 PRINT ATN (X)
RUN
? 3
 1.24905
Ok
```

3.4. CDBL

Format: CDBL(X)

Action: Converts X to a double precision number.

Example:
```
10 A = 454.67
20 PRINT A; CDBL(A)
RUN
 454.67    454.6700134277344
Ok
```

3.5 CHR$

Format: CHR$(I)

Action: Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix E.) CHR$ is commonly used to send a special character to the terminal.

Special characters are:

CHR$(4);CHR$(i);CHR$(j);	set cursor on position j in line i CHR$(5); print screen contents on printer
CHR$(7);	bell
CHR$(12);	clear screen, cursor home

Example:

```
A$=CHR$(12)+CHR$(7)+CHR$(4)+CHR$(8)+CHR$(25)
PRINT A$;"XYZ"
clears screen, rings the bell
cursor to line 8, position 25
XYZ on positions 25, 26 and 27
cursor to begin line 9
```

3.6 CINT

Format: CINT(X)

Action: Converts X to an integer by rounding the fractional portion. If X is not in the range -32768 to 32767, an "Overflow" error occurs.

Example:

```
PRINT CINT(45.67)
 46
Ok
```

See the CDBL and CSNG functions for converting numbers to the double precision and single precision data type. See also the FIX and INT functions, both of which return integers.

3.7 COS

Format: COS(X)

Action: Returns the cosine of X in radians. The calculation of COS(X) is performed in single precision.

Example:
```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

NOTE: To calculate π and to convert degrees into radians v.v. the next construction can be used.

```
10 PI = 4 * ATN(1)
20 RAD = 180 / PI
30 LPRINT PI
40 LPRINT RAD
50 LPRINT " 1.5 RADIANS = ";1.5 * RAD;"DEGREES"
60 LPRINT " 45  DEGREES = ";45  / RAD;"RADIANS"
RUN
 3.14159
 57.2958
 1.5 RADIANS = 85.9437 DEGREES
 45  DEGREES = .785398 RADIANS
```

3.8 CSNG

Format: CSNG(X)

Action: Converts X to a single precision number.

Example:
```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
RUN
 975.3421  975.342
Ok
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types.

3.9 CVI, CVS, CVD

Format:

```
CVI(<2-byte string>)
CVS(<4-byte string>)
CVD(<8-byte string>)
```

Action: Convert string values to numeric values. Numeric value that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example:

```
.
.
.
70 FIELD #1,4 A$ N$, 12 AS B$, ...
80 GET #1
90 Y=CVS(N$)
.
.
.
```

See also MKI$, MKS$, Appendix B.

3.10 EOF

Format:

```
EOF(<file number>)
```

Action: Returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while INPUTting, to avoid "Input past end" errors.

Example:

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
.
.
.
```

3.11 EXP

Format: EXP(X)

Action: Returns e to the power of X. X must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:
```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.5982
Ok
```

3.12 FIX

Format: FIX(X)

Action: Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

Examples:
```
PRINT FIX(58.75)
 58
Ok

PRINT FIX (-58.75)
-58
Ok
```

3.13 FRE

Format:
```
FRE(0)
FRE("")
```

Action: Arguments to FRE are dummy arguments. FRE (0) returns the number of bytes in memory that are not being used by BASIC.

FRE("") forces a garbage collection before returning the number of free bytes. BE PATIENT: garbage collection may take 1 to 1-1/2 minutes. BASIC will not initiate garbage collection until all free memory has been used up.

Examples:
```
PRINT FRE(0)
 20838
Ok
```

3.14 HEX$

Format:
```
HEX$(X)
```

Action: Returns a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX$(X) is evaluated.

Example:
```
10  INPUT X
20  A$ = HEX$(X)
30  PRINT X "DECIMAL IS " A$ " HEXADECIMAL "
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

See the OCT$ function for octal conversion.

3.15 INP

Format: INP(I)

Action: Returns the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement.

Example:
```
100 A=INP(255)
```

3.16 INPUT

Format: INPUT$(X[,[#]Y])

Action: returns a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed and all control characters are passed through except SHIFT-STOP, which is used to interrupt the execution of the INPUT$ function.

Example 1:
```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN
HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

Example 2:
```
  .
  .
  .
100 PRINT "TYPE P TO PRECEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
  .
  .
  .
```

3.17 INSTR

Format: INSTR([I,]X$,Y$)

Action: Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in the range 0 to 255. If I LEN(X$) or if X$ is null or if Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions or string literals.

Example:
```
10 X$  =  "ABCDEB"
20 Y$  =  "B"
30 PRINT INSTR(X$,Y$); INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

3.18 INT

Format: INT(X)

Action: Returns the largest integer <=X.

Examples:
```
PRINT INT (99.89)
  99
Ok

PRINT INT(-12.11)
-13
Ok
```

See the FIX and CINT functions which also return integer values.

3.19 LEFT$

Format: LEFT$(X$,I)

Action: Returns a string comprised of the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than LEN (X$), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

Example:
```
10 A$ = "BASIC-80"
20 B$ = LEFT$(A$,5)
30 PRINT B$
BASIC
Ok
```

Also see the MID$ and RIGHT$ functions.

3.20 LEN

Format: LEN(X$)

Action: Returns the number of characters in X$. Non-printing characters and blanks are counted.

Example:
```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
 16
Ok
```

3.21 LOC

Format: LOC(<file number>)

Action: With random disk files, LOC returns the next record number to be used if a GET or PUT (without a record number) is executed. With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was OPENed.

Example:
```
200 IF LOC(1)>50 THEN STOP
```

3.22 LOG

Format: LOG(X)

Action: Returns the natural logarithm of X. X must be greater than zero.

Example:
```
PRINT LOG(45/7)
 1.86075
Ok
```

3.23 LPOS

Format: LPOS(X)

Action: Returns the current position of the line printer print head within the line printer buffer. Does not necessarily give the physical position of the print head. X is a dummy argument.

Example:
```
100 IF LPOS (X)>60 THEN LPRINT
```

3.24 MID$

Format: MID$(X$,I[,J])

Action: Returns a string of length J characters form X$ beginning with the Ith character. I and J must be in the range 0 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character all rightmost characters beginning with the Ith character are returned.
If I>LEN(X$), MID$ returns a null string.

Example:
```
LIST
10 A$="GOOD"
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$; MID$(B$,9,7)
Ok
RUN
GOOD EVENING
Ok
```

Also see the LEFT$ and RIGHT$ functions.

3.25 MKI$, MKS$, MKD$

Format

```
MKI$(<integer expression>)
MKS$(<single precision expression>)
MKD$(<double precision expression>)
```

Action: Convert numeric values to string values. Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Example:

```
90 AMT=(K+T)
100 FIELD #1. 8AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
  .
  .
```

See also CVI, CVS, CVD, Appendix B.

3.26 OCT$

Format: OCT$(X)

Action: Returns a string which represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example:

```
PRINT OCT$(24)
 30
Ok
```

See the HEX$ function for hexadecimal conversion.

3.27 PEEK

Format: PEEK(I)

Action: Returns the byte (decimal integer in the range 0 to 255.) read from memory location I. I must be in the range 0 to 65536, PEEK is the complementary function to the POKE statement, Chapter 2.

Example: A=PEEK(&H5AOO)

3.28 POS

Format: POS(I)

Action: Returns the current cursor position. The leftmost position is 1. X is a dummy argument.

Example: IF POS(X)<60 THEN PRINT

Also see the LPOS function.

3.29 RIGHT$

Format: RIGHT$(X$,I)

Action: Returns the rightmost I characters of string X$. If I=LEN(X$), returns X$. If I=0, the null string (length zero) is returned.

Example:
```
10 A$="DISK BASIC-80"
20 PRINT RIGHT$(A$,8)
RUN
BASIC-80
Ok
```

Also see the MID$ and LEFT$ functions.

3.30 RND

Format: RND[(X)]

Action: Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see RANDOMIZE, chapter 2. However, x<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example:
```
10    FOR I=1 TO 5
20    PRINT INT(RND*100);
30    NEXT
RUN
 24   30   31   51   5
Ok
```

3.31 SGN

Format: SGN(X)

Action: If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example: ON SGN(X)+2 GOTO 100, 200, 300 branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.

3.32 SIN

Format: SIN(X)

Action: Returns the sine of X in radians. SIN (X) is calculated in single precision.

Example:
```
PRINT SIN (1.5)
 .997495
Ok
```

3.33 SPACE$

Format: SPACE$(X)

Action: Returns a string of spaces of length X. The expressions X is rounded to an integer and must be in the range 0 to 255.

Example:
```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
   2
    3
     4
      5
 Ok
```

Also see the SPC function.

3.34 SPC

Format: SPC(I)

Action: Prints I blanks on the terminal. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255.

Example:
```
PRINT "OVER" SPC(15) " THERE"
OVER                 THERE
Ok
```

Also see the SPACE$ function.

3.35 SQR

Format: SQR(X)

Action: Returns the square root of X. X must be >=0, otherwise an "Illegal function call" error occurs.

Example:
```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10                3.16228
 15                3.87298
 20                4.47214
 25                5
Ok
```

3.36 STR$

Format: STR$(X)

Action: Returns a string representation of the value of X.

Example:
```
5  REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
    .
    .
    .
```

Also see the VAL function.

3.37 STRING$

Formats:
```
STRING$ (I,J)
STRING$ (I,X$)
```

Action: Return a string of length I whose characters all have ASCII code J or the first character of X$.

Example:
```
10 X$ = STRINGS$ (10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

3.38 TAB

Format:
```
TAB(I)
```

Action: Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

Example:
```
10  PRINT "NAME" TAB(25) "AMOUNT": PRINT
20  READ A$,B$
30  PRINT A$ TAB(25) B$
40  DATA "G. T. JONES", "$25.OO"
RUN
NAME                    AMOUNT

G. T. JONES             $ 25.00
Ok
```

3.39 TAN

Format: TAN(X)

Action: Returns the tangent of X in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:
```
10 Y = Q*TAN(X)/2
```

3.40 USR

Format: USR[<digit>](X)

Action: Calls the user's assembly language subroutine with the argument X. <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See Appendix A.

Example:
```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
   .
   .
   .
```

3.41 VAL

Format: VAL(X$)

Action: Returns the numerical value of string X$. If the first non-blank character of X$ is not +, -, &, or a digit, VAL(X$)=0.

Example:

```
10 X$="-123"
20 PRINT VAL(X$)
RUN
  -123
    .
    .
    .
50 X$="  15 PETER"
60 PRINT VAL(X$)
RUN
    15
    .
    .
    .
100 X$= "PETER 15"
110 PRINT VAL(X$)
RUN
     0
    .
    .
    .
```

See the STR$ function for numeric to string conversion.

APPENDIX A

BASIC File Commands and Functions

A.1 SYSTEM COMMAND

Format: SYSTEM A or SYSTEM B

Purpose: To set the default drive to the specified drive number.

A.2 DISK FILES

Disk filenames follow the normal BASIC naming conventions. All filenames may include A: or B: as the first two characters to specify a disk drive, otherwise the currently selected drive is assumed. A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN filename commands if no "." appears in the filename and the filename is less than 9 characters long.

A.3 FILES COMMAND

Format: FILES[filename]

Purpose: To print the names of files residing on the current disk.

Remarks: If filename is omitted, all the files on the currently selected drive will be listed.
filename is a string formula which may contain question marks (?) to match any character in the filename or extension. An Asterisk (*) as the first character of the filename or extension will match any file or any extension.
Be patient: It only few files are present on the assigned disk, the FILES command can be very slow.

Examples:
```
FILES
FILES "*BAS"
FILES "B:*.*"
FILES "TEST?.BAS"
```

A.4 RESET COMMAND

Format: RESET

Purpose: To tell the system that an other diskette has been inserted.

Remarks: Before exchanging a diskette, all files have to be closed. Then the disk administration is updated.
After exchanging, the RESET statement has to be given. The system reads then the administration of the new inserted diskette.

Note: After DISK I/O ERROR the statement RESET has to be given.

A.5 LOF FUNCTION

Format: LOF(file number)

Action: Returns the number of records present in the last extent read or written. If the file does not exceed one extent (64 records), then LOF returns the true length of the file.

Example:
```
110 PRINT "MY CURRENT RECORD NUMBER ON FILE 1
IS", LOF(1)
```

A.6 EOF

With BASIC the EOF function may be used with random files. If a GET is done past the end of file, EOF will return -1. This may be used to find the size of a file using a binary search or other algorithm.

APPENDIX B

BASIC Disk I/O

Disk I/O procedures for the beginning BASIC user are examined in this appendix. If you are new to BASIC or if you're getting disk related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to the BASIC system's requirements for filenames. The file system will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

B.1 PROGRAM FILE COMMANDS

Here is a review of the commands and statements used in program file manipulation.

SAVE "filename"[,A]	Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)
LOAD "filename"[,R]	Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files.

RUN "filename"[,R]	RUN "filename" loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.
MERGE "filename"	Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.
KILL"filename"	Deletes the file from the disk. "filename" may be a program file, or a sequential or random access data file.
NAME	To change the name of a disk file, execute the NAME statement, NAME "oldfile" AS "newfile". NAME may be used with program files, random files, or sequential files.

B.2 PROTECTED FILES

If you wish to save a program in an encoded binary format. use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. The attempt to list or edit the protected program results in an "illegal function call".

B.3 DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a BASIC program: sequential files and random access files.

B.3.1 Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

```
OPEN    PRINT#          INPUT#        WRITE#
        PRINT# USING    LINE INPUT#

CLOSE   EOF     LOC
```

The following program steps are required to create a sequential file and access the data in the file:

1.	OPEN the file in "O" mode.	`OPEN "O",#1,"DATA"`
2.	Write data to the file using the PRINT# statement. (WRITE# maybe used instead.)	`PRINT#1,A$;B$;C$`
3.	To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.	`CLOSE#1` `OPEN "I",#1,"DATA"`
4.	Use the INPUT# statement to read data from the sequential file into the program.	`INPUT#1,X$,Y$,Z$`

Program B-1 is a short program that creates a sequential file, "DATA", from information you input at the terminal.

```
10 OPEN "O",#1,"DATA2
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN


NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

PROGRAM B-1 - CREATE A SEQUENTIAL DATA FILE

Now look at Program B-2. It accesses the file "DATA" that was created in Program B-1 and displays the name of everyone hired in 1978.

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

PROGRAM B-2 - ACCESSING A SEQUENTIAL FILE

Program B-2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. A sector is a 256-byte block of data.

B.3.1.1 Adding Data To A Sequential File -

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. The following procedure can be used to add data to an existing file called "NAMES".

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY".
4. CLOSE "NAMES" and KILL it.
5. Write the new information to "COPY".
6. Rename "COPY" as "NAMES" and CLOSE.
7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program B-3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO COPY
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

PROGRAM B-3 - ADDING DATA TO A SEQUENTIAL FILE

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

B.3.2 Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk -- it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

OPEN	FIELD	LSET/RSET	GET
PUT	CLOSE	LOC	
MKI$	CVI		
MKS$	CVS		
MKD$	CVD		

B.3.2.1 Creating A Random File -
The following program steps are required to create a random file.

1. Open the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 256 bytes.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

```
FIELD #1 20 AS N$,
 4 AS A$, 8 AS P$
```

3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.

```
LSET N$=X$
LSET A$=MKS$(AMT)
LSET P$=TEL$
```

4. Write the data from the buffer to the disk using the PUT statement.

```
PUT #1,CODE%
```

Look at Program B-4. It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

NOTE

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R",#1,"FILE
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

PROGRAM B-4 - CREATE A RANDOM FILE

B.3.2.2 Access A Random File -
The following program steps are required to access a random file:

1. Open the file in "R" mode.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

```
FIELD #1 20 AS N$
         4 AS A$, 8 AS P$
```

NOTE:
In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.

```
GET #1,CODE%
```

4. The data in the buffer may now be acessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precission values, and CVD for double precision values.

```
PRINT N$
PRINT CVSD(A$)
```

Program B-5 accesses the random file "FILE" that was created in Program B-4. By inputting the three-digit code at the terminal, the information associated with that code is read from the file and displayed.

```
10 INPUT "R",#1,"FILE"
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

PROGRAM B-5 -ACCESS A RANDOM FILE

The LOC function, with random files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement.
For example, the statement

```
IF LOC(1)=50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

Program B-6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

PROGRAM B-6 - INVENTORY

```
120 OPEN "R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPlAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF~(FUNCTION 1)OR(FUNCTION>6) THEN PRINT "BAD FUNCTION
    NUMBER":GOTO 130
230 IF FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)~ 255 THEN INPUT"OVERWRITE";A$:IF A$  "Y" THEN
    RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
```

```
600 INPUT "QUANTITY TO SUBSTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%) 0 THEN PRINT "ONLY";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-S%
640 UD Q%= CVI(R$) THEN PRINT "QUANTITY NOW";Q%;;" REORDER
LEVEL"+CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$) CVI(R$) THEN PRINT D$;" QUANTITY"; CVI(Q$)
    TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%_1)OR(PART%`100) THEN PRINT "BAD PART NUMBER":GOTO
    840
890 END
900 REM INITIALIZE
910 INPUT "ARE YOU SURE":IF B$  "Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

APPENDIX C

Assembly Language Subroutines

BASIC has provisions for interfacing with assembly language subroutines. The USR Function allows assembly language subroutines to be called in the same way BASIC's intrinsic functions are called.

C.1 MEMORY ALLOCATION

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). BASIC uses all memory available from its starting location up, so only the topmost locations in memory can be set aside for user subroutines.

When an assembly language subroutine is called, the stack pointer is set up for levels (16 bytes) of stack storage. If more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

The assembly language subroutine may be loaded into memory by means of the BASIC POKE statement.

C.2 USER FUNCTION CALLS

The format of the USR function is

USR[<digit>](argument)

where <digit> is from 0 to 9 and the argument is any numeric or string expression. <digit> specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A may be one of the following:

Value in A	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating point number
8	Double precision floating point number

If the argument is a number, the [H,L] register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC-3 contains the lower 8 bits of the argument and
FAC-2 contains the upper 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-3 contains the lowest 8 bits of mantissa and
FAC-2 contains the middle 8 bits of mantissa and
FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).
FAC-0 is the exponent minus 128, and the binary point is to the left of the most significant byte of the mantissa.

If the argument is a double precision floating point number:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bytes).

If the argument is a string, the [D,E] register pair points to 3 bytes called the "string descriptor." Byte o of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

CAUTION: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

```
A$ = "BASIC"+""
```

This will copy the string literal into string space and will prevent alteration of program text during a subroutine call.

C.3 CALL STATEMENT

BASIC user function calls may also be made with the CALL statement.

A CALL statement with no arguments generates a simple "CALL" instruction. The corresponding subroutine should return via a simple "RET."

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. that parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).

2. If the number of parameters is greater than 3, they are passed as follows:

    1. Parameter in HL.

    2. Parameter in DE.

    3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that, with this scheme, the subroutine must know how many parameters to exprect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for correct number or type of parameters.

When accessing parameters in a subroutine, don't forget that they are pointers to the actual arguments passed.

NOTE

It is entirely up to the programmer to see to it that the arguments in the calling program match in number, type, and length with the parameters expected by the subroutine. This applies to BASIC subroutines, as well as those written in assembly language.

APPENDIX D

Mathematical Function

Derived Functions

Functions that are not intrinsic to BASIC may be calculated as follows:

Function	BASIC Equivalent
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN(X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1)) +((SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=EXP(-X)/EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP)-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

APPENDIX E

ASCII Character Codes

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	
007	BEL	050	2	093	]
008	BS	051	3	094	
009	HT	052	4	095	
010	LF	053	5	096	'
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060		103	g
018	DC2	061	=	104	h
019	DC3	062		105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
301	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	
038	&	081	Q	124	
039	'	082	R	125	
040	(	083	S	126	
041	)	084	T	127	DEL
042	*	085	U		

ASCII codes are in decimal
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

APPENDIX F

Converting BASIC programs

If you have programs written in other BASIC's than Philips-BASIC, some minor adjustments may be necessary before running them with BASIC. Here are some specific things to look for when converting BASIC programs.

F.1 STRING DIMENSIONS

In some old BASIC implementations the array elements of a string variable contained only one character, so one of the dimensions was used to specify the length of a string. Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the BASIC statement DIM A$(J).

In BASIC, the MID$, RIGHT$ and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Old BASICs	Philips-BASIC
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$(J),I,1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Old BASICs	Philips-BASIC
A$(I)=X$	MID$(A$,1,1,)=X$
A$(I,J)=X$	MID$(A$(J),I,1)=X$

Some BASIC's use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for BASIC string concatenation.

F.2 MULTIPLE ASSIGNMENTS

Some BASIC's allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

F.3 MULTIPLE STATEMENTS

Some BASIC's use a backslash (\) to separate multiple statements on a line. With BASIC, be sure all statements on a line are separated by a colon (:).

F.4 MAT FUNCTIONS

Programs using the MAT functions available in some BASIC's must be rewritten with FOR...NEXT loops to execute properly.

APPENDIX G

Summary of Error Codes and Error Messages

Number	Message
1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.)
3	Return without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	Illegal function call A parameter that is out of range is passed to math or string function. An FC error may also occur as the result off: 1. a negative or unreasonably large subscript 2. a negative or zero argument with LOG 3. a negative argument to SQR 4. a negative mantissa with a non-integer exponent

5. a call to a USR function for which the starting address has not yet been given

6. an inproper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.

6 Overflow
The result of a calculation is too large to be represented in BASIC number format. If underflow occurs, the result is zero and execution continues without an error.

7 Out of memory
A program is too large, has too many FOR loops or GOSUB's, too many variables, or expressions that are too complicated.

8 Undefined line number
A line reference in a GOTO, GOSUB, IF...THEN...ElSE or DELETE is to a nonexistent line.

9 Subscript out of range
An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

10 Duplicate Definition
Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.

11 Division by zero
A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

12 Illegal direct
A statement that is illegal in direct mode is entered as a direct mode command.

13 Type mismatch
A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

14 Out of string space
String variables exceed the amount of allocated string space. BASIC will allocate 50 bytes of stringsspace, unless string space is allocated by the CLEAR statement.

15 String too long
An attempt is made to create a string more than 255 characters long.

16 String formula too complex
A string expression is too long or too complex. The expression should be broken into smaller expressions.

17 Can't continue
An attempt is made to continue a program that:

1. has halted due to an error,

2. has been modified during a break in execution, or

3. does not exist.

18 Undefined user function
A USR function is called before the function definition (DEF statement) is given.

19 No RESUME
An error trapping routine is entered but contains no RESUME statement.

20 RESUME without error
A RESUME statement is encountered before an error trapping routine is entered.

21 Unprintable error
An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

22 Missing operand
An expression contains an operator with no operand following it.

23 Line buffer overflow
An attempt is made to input a line that has too many characters.

26 FOR without NEXT
A FOR was ecountered without a matching NEXT.

29 WHILE without WEND
A WHILE statement does not have a matching WEND

30 WEND without WHILE
A WEND was ecountered without a matching WHILE.

31 Printer error

50 Field overflow
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

51 Internal error
An internal malfunction has occurred in Philips BASIC. Report to Philips under which the message appeared.

52 Bad file number
A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

53 File not found
A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.

54 Bad file mode
An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O, or R.

55 File already open
A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.

57 Disk I/O error
An I/O error occured on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error

58 File already exists
The filename specified in a NAME statement is

61 Disk full
All disk storage space is in use.

62 Input past end
An INPUT statement is executed after all the data in the file has been INPUT, or for a null [empty] file. To avoid this error, use the EOF function to detect the end of file.

63 Bad record number
In a PUT or GET statement, the record number is either greater than the maximum allowed [32767] or equal to zero.

64 Bad file name
An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN [e.g., a filname with too many characters].

66 Direct statement in file
A direct statement is encountered while while LOADing an ASCII-format file. The LOAD is terminated.

67 Too many files
An attempt is made to create a new file [using SAVE or OPEN] when all 255 directory entries are full.

APPENDIX H

The Volume Organization Utility

The Volume Organization Utility supports the field-development of BASIC user-programs.

The utility is invoked by:

```
RUN  "B:VOLORG"
```

and the user is asked to select the desired function via a menue.

The functions are:

0. load and run a BASIC program
1. read disk directory
2. copy file
3. copy disk from drive 1 to drive 2
4. delete a file
5. delete whole disk
6. rename a file
7. set/reset file write-protection
8. display file sizes
9. compare disks
A. set parameters for automatic program load
E. exit to BASIC command level

After having selected a function, the utility prompts for all necessary inputs.
The file names follow the normal BASIC conventions, except that drive A: is referred to by 1 and drive B: by 2.

After a function has been executed successfully, the utility asks for continuation.

Responding with "1" simply executes the same function again, while any other key returns to the menue. After an unsuccessful termination of a function any key returns to the menue.

The disk directory can be printed with function 1 when the user types "P" as a response to the question "NEXT PAGE".

With the function "A" parameters for the systems start-up time can be set:

> The file name of a BASIC program which is RUN after the start-up,
> the number of files which can be opened at the same time (0 through 15), and whether an index sequential excess method (ISAM) is required or not (yes/no)

APPENDIX I

Screen I/O Management

BASIC has provisions for the Screen's I/O Management. These procedures are examined in this appendix.

Be careful: The incorrect use of memory address, as mentioned hereinafter, may cause unpredictable results.

I.1 Clear Screen, Cursor Home

With PRINT CHR$(12); the screen is cleared, and the cursor goes to the home position

I.2 Ring The Bell

With PRINT CHR$(7); the bell rings

I.3 Move Cursor

To move the cursor to line , position
execute PRINT CHR$(4); CHR$(line), CHR$(position)

The allowable range for line is 1 through 24,
for position 1 through 80

I.4 Move Cursor To The Beginning Of The Next Line

PRINT

I.5 Print Screen

The screen's image is routed to the printer by
PRINT CHR$(5);

I.6 Change Screen Output Speed

At power-up or after a system reset the screen is set to the highest output speed, which is indicated by a 0 (Zero byte) on location &H6090.

You can slow down the output speed by resetting it with e.g. POKE &H6090, 8.

The output speed of the screen slows down proportionally to this value until another POKE to the address is executed. The lowest possible output speed is set by POKE &H6090, 255.

I.7 The Keyboard Input Conversion Table

The input which is entered by the user via the keyboard is translated to ASCII characters by the keyboard input conversion table. In the case, that the user wants to disable, or to change the meaning of a key, he or she simply alters the contents of the conversion table.

The ASCII characters which are read in from the keyboard queue after you strike a key, are stored in the memory locations A to A + &H9F in which A can be calculated as follows:

```
A = 256*PEEK (&H6939) + PEEK (&H6938)
```

In this table the ASCII code can be found on location:
A + Keyboard position code

Example 1:

The command POKE A + &H51, ASC("Y") changes the meaning of the key (upper case) "H". An ASCII "Y" (hexadecimal 59) will appear on the screen and be recognized from your BASIC program, if you strike the "H" key, until this location of the keyboard input table is altered again.

Example 2:

The user can disable the STOP key e.g. by replacing it by an ASCII space (hexadecimal 20):

```
POKE A + &H58, ASC (" ")
```

APPENDIX J

BASIC Reserved Words

ABS
AND
ASC
ATN
AUTO
CALL
CDBL
CHAIN
CHR$
CINT
CLEAR
CLOSE
COMMON
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
EDIT
ELSE
END
EQV
ERASE
ERL
ERR
ERROR
EXP
FIELD
FILES
FIX
FN
FOR
FRE
GET
GO TO
GOSUB
GOTO
HEX$
IF
IMP
INKEY
INKEY$
INP
INPUT
INSTR
INT
KILL
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOF
LOG
LPOS
LPRINT
LSET
MERGE
MID$
MKD$
MKI$
MKS$
MOD
NAME
NEW
NEXT
NOT
NULL
OCT$
ON
OPEN
OPTION
OR
OUT
PEEK
POKE
POS
PRINT
PUT
RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SGN
SIN
SPACE$
SPC
SQR
STEP
STOP
STR$
STRING$
SWAP
SYSTEM
TAB
TAN
THEN
TO
TROFF
TRON
USING
USR
VAL
VARPTR
WAIT
WEND
WHILE
WIDTH
WRITE
XOR