USA

Panasonic Company
Division of Matsushita Electric Corporation of America
One Panasonic Way,
Secaucus, New Jersey 07094

Panasonic Hawaii Inc.
91-238 Kauhi St. Ewa Beach
P.O. Box 774
Honolulu, Hawaii 96808-0774

Panasonic Sales Company
Division of Matsushita Electric of Puerto Rico, Inc.
Ave. 65 De Infanteria, KM 9.7
Victoria Industrial Park
Carolina, Puerto Rico 00630

CANADA

Panasonic Canada
Division of Matsushita Electric of Canada Limited
5770 Ambler Drive, Mississaugo,
Ontario L4W2T3

OTHERS

Matsushita Electric Trading Co., Ltd.
32nd floor, World Trade Center Bldg.,
No. 4-1, Hamamatsu-Cho 2-Chome,
Minato-Ku, Tokyo 105, Japan
Tokyo Branch P.O. Box 18 Trade Center

Panasonic

HHC T.M.

SnapBASIC

VOLUME II: REFERENCE GUIDE

AN ADVANCED PROGRAMMING LANGUAGE FOR THE HHC™

RL-S6002S8

SnapBASIC

an advanced programming language for the HHC™

VOLUME II: REFERENCE GUIDE

TABLE OF CONTENTS

CHAPTER 8: PEEKS AND POKES

CHAPTER 1: STATEMENTS AND COMMANDS

Below is a complete description of every statement and command that SnapBASIC accepts. In describing the syntax of the statements and commands, we will use the following notation:

- Upper case letters should be used as shown.
- Lower case letters are "placeholders" for something you must fill in when you write the statement.
- '[x]' means "x is optional."
- '[x|y|z]' means "you may choose one of x or y or z, or you may omit this expression."
- ' {x|y|z }' means "you must choose one of x or y or z."

exp	represents an expression
ln	represents a line number
lnexp	is an expression, representing a line number, or simply a line number
n	represents a numeric value (constant, variable, expression, etc).[1]
nv	represents a numeric variable or array element
nc	represents a numeric constant
offset	represents a byte offset between 0 and flen(0)-1
recnr	represents a record number between 0 and frec(0)-1
s	represents a string value
sv	indicates a string variable
sc	represents a string constant, e.g. "Type name:"
v	represents a variable of either numeric or string type
x	represents a value of either numeric or string type

Where we need more than one element of the same type, we will add numbers to the names: ***e.g.***, n1, n2, n3, . . .

[1] - Integer and numeric values are interchangeable, except where noted otherwise.

?

See ***PRINT***. '?' is an abbreviation for PRINT.

Examples:

```
? Miles
? "Fuel efficiency=";Miles;"mpg."
? "Order= ;LEN (OS$)-3
? I;
? #2,I;
```

Assignment

Format: [LET] nv = n
or [LET] sv = s

The value of the expression ***n*** or ***s*** is assigned to the variable or array element ***nv*** or ***sv$***. Type conversion is done automatically, according to a conversion table.

The reserved word LET is allowed for compatibility with some other versions of BASIC, which require it. An assignment statement has the same effect whether or not LET is used.

Examples:

```
Fuel=23
X=5
LET X=5
X=X+1
XS=5*(XA(N)^2+ABS (Y))
Warning$="All systems go"
Warning$=Warning$+"."
```

ATTACH

Format: ATTACH n1 TO #n2

Attaches a device with device code ***n1*** to LUN (logical unit number) ***n2*** and turns the device ON.

Valid values of ***n1*** for each peripheral are given in Chapter 7.

Valid values for ***n2*** are 0 through 15. 0 is the system input unit, normally attached to the keyboard. 1 is the system output unit, normally attached to the LCD. 2 through 15 have no "normal" attachments; you must attach devices explicitly before you can use them.

ATTACH overrides the I/O key; a device that shows an "OFF" status in the I/O menu will show "ON" after an ATTACH.

Notes:

1. The typeahead feature is active as long as you do not change input units. When the input unit number is changed, SnapBASIC's typeahead buffer is cleared. Characters in peripheral's buffers are not cleared.
2. The CLEAR key does ***not*** affect the ATTACHed or DETACHed status of LUN's. If a LUN was already attached, a subsequent ATTACH will first DETACH the previously attached LUN.

Examples:

```
ATTACH 68 TO #2
ATTACH UN TO #2+BA
ATTACH DT(N) TO #3
```

AUTO (immediate mode only)

Format: AUTO n1,n2

AUTO provides automatic insertion of program line numbers; ***n1*** and ***n2*** specify the starting line number and the increment for successive line numbers, respectively. The increment may also be negative, in which case you can proceed backwards through your program.

When AUTO is executed, the first line number is displayed on the LCD. If the AUTO feature is used with an already existing program, any existing program line will be overwritten by the new program line. There is one exception: typing an ENTER immediately after the line number will keep the original line unchanged.

To turn off the AUTO feature, press the BREAK key.

Examples:

```
AUTO 100 , 5
AUTO 200,-5
```

BURN (immediate mode only)

Format: BURN sc

BURN is a very powerful command that transforms the current BASIC program in memory into a stand-alone capsule image file that can be "burned" into an EPROM capsule by the HHC's EPROM burner peripheral. ***sc*** is the name assigned to the capsule that is produced (maximum of 16 characters).

The BURN command removes much of the file during execution, making it unuseable from within BASIC. ***Always*** make a copy of the file before using the BURN command. After execution of BURN, the file's name is the same, but its type is changed from a SnapBASIC program file (20 Hex) to a capsule image file (02 Hex); the file no longer appears in SnapBASIC's menu, but will appear within the menu of the EPROM burner peripheral.

The EPROM burner peripheral in combination with a RAM/ROM peripheral will allow you to test the capsule image form of the program before an EPROM is actually produced. [The RAM/ROM is a special product of Friends Amis, Inc.]

Since the capsule image file contains a byte-for-byte image of the future contents of a capsule, it is also possible to transfer these contents to a standard EPROM programmer device available from a number of manufacturers. The capsule image file is transferred through the HHC's RS232C peripheral using simple utility programs that can be easily developed with the SnapBASIC or SnapFORTH ™ capsules.

While the BURN command is operating, the display shows a listing of the program lines being loaded into the capsule image. It is good practice to produce a printed listing by ATTACHing to a printer to have a more permanent form of the program. After a successful BURN, the message

```
SIZE=nnnn
```

is printed, where ***nnnn*** is the amount of EPROM space required. Press any key to continue.

If a line in your program contains a statement or function that cannot be burned, or if certain other errors occur, an error message is displayed. The error message keeps alternating with the listing of the line that caused the error and the HHC beeps. Type any key to return to the primary menu. The partially BURNed file and the unBURNed source file are both deleted. You can then copy the backup of the file back into the working RAM area and correct the error.

Caution: Do not use the CLEAR key after a BURN error, or a useless version of the partially BURNed file will remain in memory. If this happens, delete the file using the FILE SYSTEM.

BURN reduces the generated code to minimum memory requirements by:

1. Removing all REM statements
2. Merging all DATA to one large DATA statement
3. Replacing all GOTO's and GOSUB's with absolute jumps
4. Removing all line number tables, effectively destroying all line information
5. Removing all the names of all the variables
6. Loading and linking into the image run-time routines that are actually used by the current program

The following limitations apply:

BURNing requires a substantial amount of free memory space in addition to the space used by the file. You can burn a very small file in an HHC with 4K of internal memory. For more information on the memory requirements for BURNing files, see the Tutorial manual.

BURN cannot burn the following commands: HISTORY ON, HISTORY OFF, BYE, NEW, RUN, TRON, TROFF, LIST, DEL, PAREN, GOTO ⟨expression⟩, GOSUB ⟨expression⟩. Be sure to remove these commands from your program before BURN is executed, or a 'BU' error will result during BURNing.

The GOTO's and GOSUB's that refer to a calculated address have to be replaced by an appropriate 'ON . . . GOTO' construction before the BURN command can be given.

Note that the BURN command is very powerful, but that an error in your program during the 'burn' procedure will ruin your file. In all cases it is advisable to make a backup of your program.

Example:

```
BURN "capsule name"
```

Notes On the Execution of a "BURNed" Capsule

1. The capsule is stand-alone and does not require any other capsule to be present.

2. The capsule uses a file in RAM as work space.
   The name of the file is the same as the capsule name. The file will use all the available RAM, unless otherwise specified by the FREE command (advisable when the program expands other files).
3. After a STOP, END, CLEAR, or error condition, the file in RAM will be deleted, and control returns to the primary menu.
4. When an error occurs, the error code is displayed and execution waits until the user types a key.
5. The BREAK key (C1) is disabled and has no function. Pressing BREAK during an INPUT statement is the same as pressing ENTER.
6. The number of characters that you may type ahead depends on the type ahead feature of the current input device (i.e. 7 for the keyboard).
7. BURNed programs execute somewhat faster than file programs.

BYE (Cannot BURN)

Format: BYE

BYE clears the variable values, saves the program file and returns you from the SnapBASIC interpreter, which allows you to edit and run a program, to the primary menu, which allows you to choose a program and use the I/O key.

If you use the CLEAR key, the HHC's memory area will be used to save the values and the program. You can safely go back to the program after CLEAR, but in most cases there will not be enough memory to start a new program. You should always use BYE to exit SnapBASIC.

CALL

Format: CALL lnexp

CALL calls a subroutine written in 6502 machine language (***not*** written in SnapBASIC).

lnexp is the address of the subroutine's entry point in the HHC's memory. ***lnexp*** must be in the range -32768 to 32767. Negative values of ***lnexp*** represent addresses larger than 32767.

If you are familiar with 6502 machine language, SnapBASIC's use of memory and the HHC's internal operations, you can perform functions that are not otherwise available in SnapBASIC by POKEing machine language subroutines into the HHC's memory and CALLing them.

Caution! Using an invalid value for ***lnexp*** can have catastrophic effects on your program and any other programs or data in the HHC's storage.

Example:

```
CALL 7293
```

CLEAR

Format: CLEAR

The CLEAR statement does the following things:

- Sets all numeric variables to zero, all string variables to null, and all boolean variables to FALSE.
- Resets all dimensions of array type variables.
- Sets the index of any active FOR/NEXT loop to 0. This may create unusual effects when the program is CONTinued. For example, suppose you are running the following program:

  ```
  10 FOR I=1 TO 10
  20 PRINT I;
  30 NEXT I
  ```

  If you hit the break key in the middle of the program (say, after

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

  has been printed, and then enter the commands CLEAR and CONT, the program will print out

  ```
  1 2 3
  ```

  and then stop. What has happened? SnapBASIC has indeed reset the value of the variable I; but it is the middle of the loop, and knows it was supposed to execute the loop 10 times. CONT makes the loop execute three more times, with the value of I having been reset. This is an example of the rule against changing the value of a loop variable being violated by the system itself.
- The return pointer for any active GOSUB ***may*** be altered.
- RESTOREs the program's DATA statements, if any.

Note that the CLEAR statement has no connection with the HHC's CLEAR key.

CONT (immediate mode only)

Format: CONT

Use CONT to restart a program after you have interrupted it by pressing the C1 key or after it has executed a STOP. Execution continues wherever it was interrupted, which may or may not coincide with a line number or statement.

You may display and change the values of variables while the program is interrupted. You can also LIST your program (as long as you do not change it). Except for the changes you make to variable values, the program will be in the same state when you CONTinue it as it was in when it stopped.

You may ***not*** CONTinue a program after a fatal error has occurred; after you have edited the program; or before you have begun RUNning the program.

DATA (deferred mode only)

Format: DATA v1 [, v2, v3, . . .,vn]

where each ***v*** is a numeric value or string value.

Contains data which a READ statement can read. See READ for details of use.

A DATA statement may be inserted anywhere in your program; flow of control goes around it.

About String Values

If a string value contains a comma, spaces, or a quotation mark it must be enclosed in quotation marks:

```
DATA "RED, WHITE","BLACK AND BLUE"
```

A string value with leading or trailing blanks needs to be in quotes. READ normally trims the blanks when it reads a value.

A quotation mark that appears after the first character in a string value is treated as follows:

To put a quotation mark in a string, double the quotation mark and put quotation marks around the string itself. Therefore "say ""hello"" to Max" will result in 'say "hello" to Max'.

Examples:

```
DATA 5,10,3,8,12
DATA 3
DATA 3,RED,"white",BLUE
DATA -225.771,"LABEL=(,BLP)",370
```

DEF (deferred mode only)

Format: DEF FNmm(pr1,pr2,...,prn) = ex

DEF defines a function named ***FNmm***. ***mm*** may be any name. The ***prn*** parameters are real variables and are unknown outside the function expression; they are local to only one function. Functions may be redefined by a new DEF FN.

The function name must start with FN, and the function itself should return a value of type 'real', or of type 'integer'. String functions are not allowed (FNA $ ... for example).

In evaluating the expression on the right hand side of the equals sign, all variables from the program can be used.

The ***pr***s are the function's real formal parameters; there must be at least one specified. The parameter name may be any name that would be valid as the name of a numeric real variable. SnapBASIC recognizes the difference between numeric variables and formal parameters with the same name.

ex is the body of the function definition. If it contains references to ***pr***, they refer to ***pr*** as a formal parameter, not as a variable (if there is also a variable named ***pr***).

When you refer to the function later in your program, SnapBASIC does the following:

1. Calculates the value of the function reference's argument.
2. Substitutes that value for the ***pr***s wherever ***pr***s appear in ***ex***.
3. Evaluates ***ex***.
4. Returns the real value of ***ex*** as the value of the function reference.

Examples of definitions:

```
DEF FNA(X)=X^2
DEF FNB(A,B,C,D)=A+2*B+X^2*(C-D)+Q
DEF FNargument(Q)=Q*(Q+1)
DEF FNno.ofcharacters(Z)=4+LEN (ST$)
```

Examples of use:

```
PRINT FNA(XY)
PRINT X;FNA(X-2)
X=FNnumberofcharacters(0)
```

DEL (immediate mode only)

Format: DEL lnexp
or DEL lnexp1,lnexp2

Deletes line ***lnexpr*** or deletes lines from ***lnexpr1*** through ***lnexpr2***.

Note: All of the variables in your program are allocated in a specific area within HHC RAM. If a line in your program has created a variable, and is the only reference to that variable, DELeting the line will ***not*** cause SnapBASIC to deallocate the memory assigned to that variable. If your program is running short on variable space, think twice before creating a variable-- because you will not be able to gain back the memory it occupied. Rather, you may wish to create a few temporary variables, and use them repeatedly.

Examples:

```
DEL 20
DEL 20,100
```

DETACH

Format: DETACH #n

Detaches the current device from LUN (logical unit number) ***n***. See also ATTACH.

Note: Do not DETACH #0 and #1; you will disable the keyboard and the LCD.

Example:

```
DETACH #2
```

DIM

Format:	DIM xa(n1 [, n2,. . .nx])	for real array
or	DIM xa$(n1[, n2,. . .nx])	for string array
or	DIM xa?(n1[, n2,. . .nx])	for boolean array
or	DIM xa%(n1[, n2,. . .nx])	for integer array

Defines an array with ***x*** dimensions. The array has ***n1*** + ***1*** elements along its first dimension, ***n2*** + ***1*** elements along its second dimension, and so forth.

Array elements are numbered from 0; thus, the array's elements are numbered 0 to ***n1*** along the array's first dimension, and so forth.

You can define more than one array in one DIM statement. Separate the array definitions with commas. You cannot change an array's dimensions with a new DIM statement.

A DIM statement must always precede access to an array. There is no default dimension.

Examples:

```
DIM X(10)
DIM XN$(8,2)
DIM YS(N),YN$(N),YB?(N)
```

END

Format: END

END terminates execution of your program. Unlike STOP, it does not display a 'STOPPED in line ***nnnn***' message; further, it is impossible to continue with the CONT command after the END is executed.

Whether or not there is an END (or STOP) in a BURNed program, the program will automatically return to the HHC primary menu after execution. See the BURN command for more information.

Note: Unlike some BASICs, the END statement is not required at the end of a program; an END statement is assumed to follow the last line. When you execute a trace and there is no END statement in your program, the "assumed" END will appear as line number -1.

Examples:

```
END
IF X=0 THEN END
```

FDEL

Format: FDEL recnr

FDEL deletes a record with number ***recnr*** from the current file. All records following the deleted record will be shifted one to the front (record ***recnr*** + ***1*** becoming ***recnr***, and so on).

Examples:

```
FDEL 3
```

(deletes record number 3)

To delete from record 25 to the end of the file:

```
10 FOR I = 25 TO FREC(0)-1
20 FDEL 25
30 NEXT I
```

FINS

Format: FINS recnr,stringexp

FINS inserts a record before the record with number ***recnr***, in the current file, and then writes it. Records following ***recnr*** will be shifted up (i.e., the old record ***recnr*** will become ***recnr*** + ***1***, and so on.)

If ***recnr*** 〉 frec(0)-1, the record is inserted at the end of the file.

If ***recnr*** 〈 0, the record is inserted at the beginning of the file.

See the FREE(n) function to determine and/or change the available memory.

Example:

```
FINS 3,"this is a small record"
```

FOPEN

Format: FOPEN stringexp

FOPEN opens the RAM file ***stringexp***, which becomes the current file. See the description of FREE for a specification of the interaction with the HHC system.

Note: If the file does not already exist, FOPEN will create an empty text file with the given name. To check if the file already exists, you may use the function EXIST(stringexpression), which will return the value TRUE or FALSE depending upon whether the file exists or not.

To check if there is already a file currently opened, use the FADR function; FADR(0) will return 0 if there is no file opened, or a non-zero value otherwise.

Example:

```
FOPEN "new current file"
```

FOR

Format: FOR nv = n1 TO n2 [STEP n3]
or FOR nv% = n1 TO n2 [STEP n3] loop with integer index

FOR begins a FOR/NEXT loop.

The following applies for both real and for integer (= %) indices.

SnapBASIC executes the statements between 'FOR ***nv*** = . . .' and 'NEXT ***x***' with ***nv*** = ***n1***, then with ***nv*** = ***n1*** + ***n3***, then with ***nv*** = ***n1*** + 2****n3***, etc. SnapBASIC stops looping after the last pass for which ***nv***〈 = ***n2***.

If ***n3*** is absent, SnapBASIC assumes ***n3*** = 1.

If ***n3***〈0, SnapBASIC stops the loop after the last pass for which ***nv***〉 = ***n2***.

nv may be a real or integer variable, ***not*** an array element. Using an integer variable will speed up the loop.

Terminology

nv is called the ***index*** of the loop.

n1 is called the ***initial value*** of the loop.

n2 is called the ***limit*** of the loop.

n3 is called the ***step*** or ***increment*** of the loop.

Note On Starting a Loop

SnapBASIC first calculates the values of the expressions ***n1***, ***n2*** and ***n3*** and then determines the number of times the loop is to be made. If this number is greater than 32767, then an I/O error is given at runtime and the loop is not executed, even when the index is real. Trying to change the values of ***n1 n2 n3*** within the loop is ineffective and will not change the number of times the loop is done. See the discussion under CLEAR for a ramification of this.

Note On Ending a Loop

You may leave a FOR/NEXT loop by doing a GOTO if you wish. If you do, the final value of the index will be the value it had the last time through the loop. If you allow a FOR/NEXT loop to end naturally, the final value of the index will the the value it had the last time through the loop.

Note that a FOR/NEXT loop always executes at least once, even when the initial value is past the limit.

If you want to write a NEXT that terminates only the innermost loop (or if there is only one open loop it could apply to), you may omit the index completely:

```
590 FOR I=1 TO 10        begins loop
     :
     :
620 NEXT                 ends loop
```

Speed

- Using an integer variable for the index will speed up the loop (use the %-symbol).

Notes On NEXT

You may ***not*** end two or more loops at the same point by putting both of their subscripts in the same NEXT. (This is allowed by some BASICs.) The way to end complex loops is to have the ***innermost loop first***, like this:

```
450 FOR I=1 TO 10        begins outer loop
460 FOR J=3 TO 8         begins inner loop
     :
     :
530 NEXT J               ends inner loop
540 NEXT I               ends outer loop
```

No-Nos

If two or more FOR/NEXT loops are nested and they all end at the same point, you must end every loop in a NEXT statement:

```
450 FOR I=1 TO 10        begins outer loop
460 FOR J=3 TO 8         begins inner loop
     :
     :
530 NEXT I               does not end both loops
```

Avoid doing the following things when you write FOR/NEXT loops:

- Changing the limit or step after the start of the loop. (The change will not affect the execution of the loop.)
- Entering a loop by executing a GOSUB, GOTO, or IF . . . THEN instead of executing FOR. (You'll get an NF error when you execute the NEXT.)

  You may always leave a FOR/NEXT loop with a GOSUB, then RETURN into it. You may also leave a FOR/NEXT loop with a GOTO, then return to it with a GOTO, but that is not good practice; it is difficult to follow and is liable to give rise to programming errors.
- Changing the value of the index inside the loop. SnapBASIC will continue, the loop index will be changed, but SnapBASIC will not change the number of times the loop is done.
- Allowing the index to exceed 32767.

Examples:

```
FOR I=1 TO 10
FOR I%=1 TO 11 STEP 2
FOR I%=11 TO 1 STEP -2
FOR I=3.8 TO 9.11 STEP 3.3
FOR I=K(4) TO K(5) STEP L3(5)-L3(4)
```

FPUT

Format: FPUT offset,byte

FPUT writes a single byte in a non-record file. The ***offset*** must be in the range {0...flen(0)-1 }. ***byte*** is an expression that evaluates into an integer in the range 0-255 (i.e. ***byte***= BAND(expression,255)).

Example:

```
FPUT FLEN(0)-1,0
```

(puts a 0 in the last byte of the file)

FREAD

Format: FREAD recnr,sv

FREAD reads record with number ***recnr***, and stores the contents in the ***sv***. ***Recnr*** must be in the range {0...frec(0)-1 }.

Example:

```
FREAD 3,A$
```

FREVISE

Format: FREVISE offset,length

FREVISE shrinks (if ***length*** 〈 0) or expands (if ***length*** 〉 0) the current file at the specified position. For non-record files only.

When expanding at ***offset***, the byte at this position will be shifted ***length*** bytes up. See the FREE(n) function to determine the memory available for expansion.

When shrinking at ***offset***, this byte is the first one that is deleted.

Example:

```
FREVISE OFFSET,23
```

FWRITE

Format: FWRITE recnr,sv

FWRITE writes on current file on record number ***recnr***, and overwrites any old information (take care!).

FWRITE will expand the file if the old record was smaller, or shrink the file when the old record was larger than the current one.

The maximum record length that will be written is 255 bytes.

An attempt to write on a record that does not exist gives FI error. See FINS.

Example:

```
FWRITE 3,"this overwrites!!"
```

GET

Format: GET [#n,]sv

Inputs one character from LUN (logical unit number) ***n*** and assigns it to string ***sv***.

If the LUN is omitted, SnapBASIC assumes #0 (normally the keyboard).

GET does not echo the character on the LCD, as it would for INPUT.

The ENTER and BREAK keys each count as an input character like any other. So do all of the other "non-character" keys except ON, OFF, and CLEAR which have their usual functions, and SHIFT and 2nd SFT, which apply to the next key pressed as usual.

Examples:

```
GET CH$
GET #2,CH$
```

GOSUB

Format: GOSUB ln
or GOSUB exp (Cannot BURN)

Calls a subroutine; transfers control to the first statement on line number ***ln***. Executing a RETURN will return control to the next statement after the GOSUB.

Note that the use of a line number is always faster than calculating an expression during run time.

Examples:

```
GOSUB 1010
GOSUB 10 + 2*I
```

GOTO

Format: GOTO ln
or GOTO exp (Cannot BURN)

Transfers control to the first statement on line number ***ln***. GOTO can begin execution when used in immediate mode. This is useful since no variables are cleared.

Note that the use of a line number is always faster than calculating an expression during run time.

Examples:

```
GOTO 750
GOTO K + 2*AA(I%)
```

HISTORY (Cannot BURN)

Format: HISTORY ON
and HISTORY OFF

The two forms of HISTORY shown above set or clear a HISTORY switch.

When HISTORY is ***ON***, a full trace back is produced when an error condition or a STOP command is encountered from a running program. This facility types the current line number and also prints a list of all started and as yet unfinished GOSUB's and FOR's.

When HISTORY is ***OFF***, the above feature is turned off, and the normal error and STOP messages occur.

Example:

```
430 HISTORY ON
.......
.......
480 HISTORY OFF
```

HOME

Format: HOME

Outputs a standard escape sequence. On the LCD, the display is cleared and the cursor is positioned to the left. On the micro printer, several blank lines are printed. On the TV or monitor, the screen is cleared and the cursor placed in the upper left-hand corner.

Example:

```
320 HOME
```

IF

Format: a) IF logical-expression THEN lnexp
or b) IF logical-expression THEN stmt[:stmt. . .stmt]
or c) IF logical-expression GOTO lnexp
or d) IF logical-expression GOSUB lnexp [:stmt. . .stmt]

Evaluates ***logical-expression***. ***logical-expression*** is usually an expression involving a logical operator like '<' or '>=', but it may be any expression that evaluates to a numeric value; or it may be a Boolean variable.

Forms a) and c) are identical. If ***logical-expression*** has a non-zero or TRUE value, control is transferred to the statement denoted by ***lnexp***; otherwise, execution continues at the next line.

With form b), if ***logical-expression*** is TRUE, the remaining statements on the line will be executed; otherwise, they will be skipped.

Similarly, form d) will perform the GOSUB (and the remaining statements on the line, if any) if and only if the ***logical- expression*** is TRUE.

Examples:

```
IF I<10 GOTO 950
IF I>INT(I) THEN I=INT(I)+1
```

```
IF ER THEN GOSUB 2010
IF A>B GOSUB 12+I : PRINT "YES"
IF I<>INT(I) THEN PRINT ("Must
  be an integer."):GOTO 950
```

INPUT

Format: INPUT [#n,]["prompt"{;|,}]v1,v2,v3,...|

Reads a line of input from the keyboard or from a peripheral.

n is the LUN to read from. If ***n*** is omitted, SnapBASIC assumes LUN #0 (the keyboard).

prompt is a prompting message. If it is present, INPUT prompts the user with this message. If ***prompt*** is absent, INPUT prompts the user with '?'. ***prompt*** is always sent to LUN #1.

prompt must be a string ***constant***. **Warning:** If you use a string variable, INPUT will try to read into the variable.

v1, ***v2***, ***v3***,. . . are the variables to be read. Each of them may be numeric, integer, string, or Boolean.

Input reads values into the variables in the order that variables appear in the statement. A value may be terminated by a comma, space or end-of-line; except that if a value read into a string variable begins with a quotation mark, it is terminated only by a second quotation mark or end-of-line. It is possible to read a quote "; to do so use a double quote "" as explained in DATA.

If INPUT receives too few values on the keyboard, it prompts the user for additional values with '??'. If it receives too few values from a peripheral, it simply reads another record. If input receives too many values from the (last) input line or record that it reads, it discards the extras with a message 'Rest Ignored'. Note that this is different from the behavior of READ, which saves the extras for the next READ.

Boolean input reads a string and returns FALSE if the first letter is n or N (from NO), f or F (from FALSE), or a digit #0.

Examples:

```
INPUT X
INPUT X,Y,Z,ST$
INPUT #2,X
INPUT "Next reading: ";X,Y,Z
INPUT #2,"Next reading: ";X,Y,Z
```

LET

See ***Assignment***. LET is equivalent to an assignment statement.

LIST (Cannot BURN)

Format: LIST
or LIST #n [,]
or LIST lnexp [,lnexp2]
or LIST #n,lnexp [,lnexp2]

LIST will list the complete program.

LIST ***lnexp*** will list that single line and enter the edit mode.

LIST ***lnexp,lnexp*** will list those lines, and every line in between.

n is the LUN the list should go to. If ***n*** is not given, SnapBASIC assumes LUN #1 (normally the LCD).

LIST prints BASIC keywords in upper case, expands "?" to "PRINT", inserts some spaces for readability, and adds parentheses to expressions. This often makes LISTed lines longer than they were originally typed. A unique feature is built into SnapBASIC that allows you three different types of reconstructing an expression with parentheses (see PAREN statement/command).

Interacting With LIST

When you enter LIST and press RETURN, LIST displays the first line you have requested, then the second line etc.

When multiple lines are being listed and a line is too long to fit on the LCD, LIST displays its beginning, and then rotates it until the end becomes visible. To stop the rotation before the end becomes visible, press any key. If only one line is listed, you must use the ROTATE key to review a long line.

To end LIST at any point, press the BREAK key (C1 key).

LISTing a single line will make this line the current line for the editor. See Chapter 6 about how to enter, leave and update lines in the editor.

LISTing a line that does not exist will display the message '--NONE--' and make an existing line (with the next lower line number) the current line for editing.

Note On Deferred Execution

You can use LIST in a program, but when you are done executing LIST your program will end. In other words, LIST behaves as though an END statement were built into it.

Caution: If a line LISTs in inverse characters, then the line as reconstructed is too long and characters have been lost. You must shorten the line or split the line in two to replace the lost characters.

Examples:

```
LIST
```

(list entire program)

```
LIST 20
```

(list line 20 only)

```
LIST 5,30
```

(list lines 5 through 30)

LOAD (immediate mode only)

Format: LOAD filename

LOADs and compiles the file with the name ***filename*** from the current RAM bank. If the file does not exist, error FI (File Error) results. LOAD does not remove the old program in BASIC. When you are careful with line numbers, CHAINing is possible by overwriting lines with the same line number.

During LOADing, the record numbers of the source records being LOADed are displayed.

Note: There are certain constraints on files to be loaded. First, the file must contain only a line-numbered SnapBASIC program in ASCII code; no immediate mode commands are allowed (and using them may cause unpredictable results). Lines will be truncated if they are longer than 80 characters.

Example:

```
LOAD "filename"
```

NEW (immediate mode only)

Format: NEW

NEW does the following things:

- Deletes all variables.
- Deletes all the lines from the program.

Since NEW deletes all the lines from your program, and affects the program stored in the memory area as soon as it is entered, you should use the statement ***very cautiously***.

Using the NEW command is just like executing BYE, then destroying the old file, and then selecting a new file with the old name.

NEXT

Format: NEXT
or NEXT nv

Ends a FOR/NEXT loop.

nv is the index of the loop being ended. If ***nv*** is omitted, NEXT ends the innermost loop now open.

For more information, see the description of FOR.

Note: NEXT is restricted to deferred mode, with the exception that a single line may be entered in immediate mode including both a FOR and a NEXT statement. For example,

```
FOR I = 1 TO 20: PRINT I: NEXT
```

is allowable in immediate mode.

Examples:

```
NEXT
NEXT I
```

ON

Format: ON n GOTO ln1,ln2,. . .,ln***x***
or ON n GOSUB ln1,ln2,. . .,ln***x***

Evaluates ***n*** and truncates the result to the next lower integer if necessary. If ***n***⟨1 or ***n***⟩***x***, ON does nothing (***i.e.***, the next sequential statement is executed). If ***n***=1, ON does a GOTO or GOSUB to ***ln1***. If ***n***=2, ON does a GOTO or GOSUB to ***ln2***, and so forth.

Examples:

```
ON I GOTO 110,120,130
ON I GOSUB 110,120,130
```

ONERR (deferred mode only)

Format: ONERR GOTO ln
or ONERR GOTO lnexp (cannot BURN)

Allows recovery from an error situation. When an error occurs, all LOOPs are terminated and all GOSUBs are exited. The program then continues at line ***lnexp***.

Recovery from a 'BREAK' typed by the user is done once only. Hitting the BREAK key (C1) twice will stop the program, except when a new 'ONERR' is executed. In such case the new ONERR is active and will allow one recovery on a 'BREAK' typed by the user.

After an error, the error code that would have been displayed had ONERR ***not*** been active is available as two ASCII bytes at locations 852 and 853. PEEK these locations to find out what caused the error condition; the bytes represent the first and second letters respectively of a SnapBASIC error code (see Chapter 10 for information on error codes).

Examples:

```
10 ONERR GOTO 110
10 ONERR GOTO 3*A + B
1000 IF CHR$(PEEK(852))+
     CHR$(PEEK(853))="OM"
     THEN ? "YOU RAN OUT OF RAM!":END
```

PAREN (Cannot BURN)

Format: PAREN n

This switch specifies the manner in which computational priorities are stated through the use of parentheses:

For ***n***=***0***, the parentheses are reconstructed as well as possible, matching what might be entered by a typical user.

For ***n***=***1***, parentheses are added for every priority level, to help a user see the priorities.

For ***n***=***2***, the parentheses are removed as much as possible.

Example:

If the following expression has been entered:

```
1 * 2 + (3 + 4)
```

it will be reconstructed in the following way :

```
PAREN 0 : 1*2+(3+4)
PAREN 1 : (1*2)+(3+4)
PAREN 2 : 1*2+3+4
```

POKE

Format: POKE n1,n2

Stores ***n2*** in the HHC's memory at the memory address ***n1***.

n1 must be in the range -32768 to 32767. ***n2*** must be in the range 0 to 255. Negative values of ***n1*** represent addresses larger than 32767.

Caution! Since POKE stores data directly into the HHC's memory, there is no checking to prevent you from POKEing data into the wrong place. If you use POKE with improper values for ***n1*** or ***n2***, the results will be unpredictable, and can be catastrophic. See the chapter on PEEK and POKE in the ***Snap-BASIC Tutorial Guide*** for more information.

For a discussion of useful PEEK and POKE addresses, see Chapter 8.

Example:

```
POKE 535,0
```

(changes the LCD rotation speed
to the fastest possible speed)

PRINT

Format:

PRINT [#n,] [x1] {,|;} [x2] {,|;} ... [xy] [,|;]

? [#n]

'?' is an abbreviation for PRINT. If you enter '?' in a statement, SnapBASIC will display 'PRINT' when you LIST the statement. If the line is too long when expanded, information may be lost (see LIST).

PRINT writes information to the LUN (logical unit number) specified by ***n***. If ***n*** is absent, SnapBASIC assumes LUN #1 (normally the LCD).

How Values Are Positioned

Each ***x*** is a numeric or string expression. PRINT writes the values of the expressions in the order they appear in the statement.

PRINT divides the output line into zones, each 21 characters long. The first ***x*** is displayed at the beginning of the first zone.

SnapBASIC puts no spaces between them. If the first and second ***x***'s are separated by a comma, SnapBASIC skips to the beginning of the next zone before displaying the second value.

If the last ***x*** is followed by a semicolon, SnapBASIC does not end the line; the next PRINT statement will add data to the end of the same line. If the last ***x*** is followed by a comma, SnapBA-SIC does not end the line, but skips to the start of the next zone. If the last ***x*** is followed by neither a semicolon nor a comma, SnapBASIC ends the line, so that the next PRINT statement will begin writing data at the start of the next line.

SnapBASIC has a special function, SPC$, which may be used in PRINT statement expressions. SPC$(N) inserts ***N*** spaces into the output line. In combination with the function POS, any type of tabulation can be programmed. For example, start every 12th column:

```
?1;SPC$(12-MOD(POS(0),12);2
```

SnapBASIC also has a special function, STRF$, that is able to format your floating point output.

When a PRINT is LISTed, a LUN is always shown, including LUN #1 (implied when a LUN is not specified).

PRINT With No Values

A PRINT statement may have no ***x***'s, like this:

```
PRINT
```

or

```
PRINT #2
```

Such a PRINT statement writes a blank line, or ends the current line of output if the previous PRINT statement ended with a semicolon or a comma. This form of the statement is always listed with a comma after the LUN.

Examples:

```
PRINT Miles
PRINT "Fuel efficiency=";MI;"mpg."
PRINT "Order=";LEN (OS$)-3
PRINT I
PRINT #2,I
```

READ

Format: READ v1[,v2,v3,...,vn]

READ reads data from one or more DATA statements.

Data items are read from DATA statements left-to-right and from the beginning of the program to the end. One item is read for each of the variables v1,v2,v3,...,vn.

If a READ runs out of data in one DATA statement, it begins reading the next. If a READ has data left over when it is done reading into its variables, it leaves the data for the next READ to get.

If there is no more data, READ will stop with an out-of-data error.

Examples:

```
READ Number
READ MN$(N)
READ J,SR$(J)
```

REM

Format: REM followed by a remark of any sort.

REM begins a remark line. A remark line may be used to include any sort of useful information in a program, such as the name of the author, purposes of the variables, or notes about how the program works.

SnapBASIC stores the remark line exactly as you type it. Thus, if you enter this,

```
2020 rem values within limits?
```

SnapBASIC will LIST this:

```
2020 REM values within limits?
```

Note that REM turns an entire ***line*** into a remark. The remark is not terminated by a colon, as other SnapBASIC statements are.

Also note that the if the first word of the REM statement starts immediately after the REM, this word will be written in upper case characters.

If you enter this,

```
2020remvalues within limits?
```

SnapBASIC will LIST this:

```
2020REMVALUES within limits?
```

Examples:

```
REM If array still empty, give error
REM Input: user is prompted for data
REM Copyright 1982
REM Verify that ABS(SQR(A ^+B ^))<.5
```

RESEQ (immediate mode only)

Format: RESEQ n

"Resequences" all line numbers in a program to multiples of ***n***. During the resequencing operation, the program is listed with the new line numbers. It is permissable to use the BREAK key to stop the renumbering process; in this case, only a part of the program will have been renumbered (and not all of the updates will have been made). CONT will continue the process.

Example:

```
RESEQ 20
```

(changes lines numbers to 20, 40, 60, ... etc.)

RESTORE

Format: RESTORE

"Rewinds" the program's DATA statements so that next READ statement will read the first data item on the first DATA statement.

RETURN

Format: RETURN

Returns control from a subroutine to the statement after the most recent GOSUB not yet RETURNed from.

If BASIC is not executing a subroutine, an attempt to execute a RETURN statement will cause an RT error.

If a FOR loop has been started, but is unfinished when a RETURN is executed, SnapBASIC will pop (throw away) the unfinished FOR.

RUN (cannot BURN)

Format: RUN [ln]

RUNs the current program, clearing all the variables and arrays before starting. If a line number is specified, execution begins at that line; otherwise, execution begins at the first line in the program.

If you execute RUN from within your program, the program will be restarted from the beginning; this is not generally a good idea (and can lead to an infinite loop and other anomalous results).

Examples:

```
RUN
```

(starts at the beginning)

```
RUN 30
```

(starts at line 30)

SQUEAK

Format: SQUEAK n1,n2

SQUEAKs the beeper on the HHC at pitch ***n1*** for time ***n2***.

n1 is converted into an integer. 36 gives the highest note; 0 gives the sound of silence. 32 is approximately middle C. A difference of 1 is approximately 1 semitone.

n2 is converted to an integer. The time unit is about 5.88 msec. There are 10,200 counts/minute, or 170 counts/second.

Note that a very long value for the time will cause a very long squeak, and that you cannot use the BREAK key to get out of the squeak. For example, SQUEAK 35,-1 will squeak for more than 6 minutes--and you could only stop it with the CLEAR key (which is likely to foul up the program).

Negative pitch values will give an interesting assortment of ticks, squeaks, and clicks. Values greater than 36 are reduced to the range 25 to 36 (giving an octave range of twelve semitones).

Example:

```
10 FOR I=24 TO 36
20 SQUEAK I,10
30 NEXT I
```

(makes the HHC play a chromatic scale)

STOP

Format: STOP

Interrupts execution of the program and displays the message

```
STOPPED in line nnnn
```

on the LCD. (***nnnn*** is the Line number of the line the STOP is on.)

After STOPping a program, you can continue it with the CONT statement. See CONT for more information.

TROFF (Cannot BURN)

Format: TROFF

Turns off the program execution trace that is turned on by TRON.

TRON (Cannot BURN)

Format: TRON

Turns on SnapBASIC's execution trace facility. When the trace facility is on, SnapBASIC displays the line number of each statement that it executes in deferred mode. The trace display looks like this:

```
#nn #nn #nn ...
```

where each ***nn*** is the line number of one line in the program.

Once the trace facility is turned on, it remains on until it is turned off with TROFF, or until you return to the SnapBASIC menu with BYE.

If a statement that is a GOTO loop, like

```
 10 GOTO 10
```

is encountered, TRON finds the problem and stops with a GO error.

If the program has no END statement, an implied END is executed whose line number is displayed in trace mode as -1.

Example:

```
 .
 .
 510 TRON
 .
 .
 580 TROFF
```

will trace every action between line 510 and 520.

CHAPTER 2: INTRINSIC FUNCTIONS

ARITHMETIC FUNCTIONS

nv = ABS (n)

Returns the absolute value of ***n***.

nv = EXP (n)

Returns the constant E (2.71828182846) raised to the power ***n***. The maximum value of ***n*** that will not produce an overflow error is 2357.84713522.

nv = FIX (n)

Deletes the fractional part of ***n***.

nv = FLOAT (n%)

Converts the integer number ***n*** into a floating point number. FLOAT can also be used as an alternative to INT: they behave identically for positive numbers; however, for negative numbers, FLOAT(***n***) = INT(***n***+1) (i.e., FLOAT returns the integer closest to zero).

nv = FREE (n)

n = 0: Returns the number of free bytes of memory available for storing and running programs. This is the size of the current memory area, minus the amount of space already occupied by programs and data.

For other values of ***n*** , tries to increase (decrease) the amount of memory available, such that ***n*** bytes of memory are free for the program. Such free space is used when:

1) adding a new line
2) adding a new variable
3) executing a DIM statement
4) adding a new string, or extending existing strings
5) (temporarily) input or output operations are done

Note that FREE(0) forces a "garbage collection" on dynamic strings. Garbage collection is automatically done at various times, and may take several seconds, causing an apparent loss of system to the user. If you wish to avoid this, you can execute a FREE(0) when the program is not expecting input or generating much output. The leftmost (undefined) blip is turned on during "garbage collection" as an indication to wait.

Caution: Never press the CLEAR key while the "garbage" blip is on (or anywhere else in SnapBASIC). Use the BYE command.

Note also that FREE allocates memory space for SnapBASIC. FREE(0) will take almost all the memory available in the RAM area -- so if HHC system devices or the file system ask for more memory, it may not be available. If you are going to use file operations that will expand the files, make sure that SnapBASIC does not allocate all of the memory, by using FREE(n). For example, to use 300 for BASIC and the rest for the file system, enter:

```
FREE(300)
```

To use 300 bytes for the file system and the rest for BASIC, enter:

```
FREE(FREE(0)-300)
```

nv = INT (n)

Returns the largest integer ***nv*** such that ***n***<=***nv***. INT (1.1) is 1; INT (1) is 1; INT (.9) is 0; INT (-.1) is -1.

nv = LN (n)

Returns the the natural (base E) log of ***n***. To obtain the base Y log of X, use the formula LN (X)/LN (Y). ***Example***: the base 10 (common) log of 7 is LN (7)/LN (10).

nv = LOG (n)

Returns the the common (base 10) log of ***n***. To obtain the base Y log of X, use the formula LOG (X)/LOG (Y).

nv = MAX (m,n)

Returns the maximum (larger) of ***m*** and ***n***.

nv = MIN (m,n)

Returns the minimum (smaller) of ***m*** and ***n***.

nv = MOD (m,n)

Returns the modulus of ***m*** and ***n***. Positive and negative values for ***m*** and ***n*** are handled as follows:

m	**n**	**MOD(m,n)**
8	3	2
–8	3	–2
8	–3	2
–8	–3	–2

Note that this is different from the MOD% function.

nv = PEEK (n)

Returns the value stored at memory address ***n***. ***n*** must be in the range -32768 to 32767. Negative numbers represent addresses larger than 32767. Values are returned in the range 0 to 255.

nv = POS (0)

Returns the length of the current output line.

If you execute POS immediately after ending a line of output or PRINTing a carriage return, POS returns the value 0. PRINTing a character adds 1 to the value POS will return.

The value of POS is affected by output on the LCD ***and*** on every peripheral. For example, if you execute the following code:

```
500 PRINT
510 PRINT #2
520 PRINT "ABC";
530 PRINT #2,"DEFG";
540 X=POS(0)
```

the value returned by POS will be 7, even though the lengths of the current output lines on the LCD and LUN #2 are 3 and 4, respectively. Thus, you must avoid doing output on one LUN while building up a line of output on another if you want POS to have meaning.

Note that '0' is a required dummy parameter.

nv = RND (n)

Returns a "random" number in the range 0<=RND(n)<1.

The first time you call RND, use a negative ***n*** (for example, RND(-PEEK(526)), a peek at the fastest changing value in the timer). This makes RND use ***n*** as a "seed" to begin generating a random number sequence. The same seed always produces the same sequence. RND returns the first number in the sequence.

On subsequent calls to RND, use 0 or a positive ***n***. This makes RND return the next random number in the sequence.

nv = ROUND (n)

Returns the arithmetic rounded value of ***n***, i.e. the whole number (without a fractional part) that is closest to ***n***.

nv = SQR (n)

Returns the square of ***n***. Equivalent to ***n***^2.

SQR(N) produces the same result as N^2, but executes more quickly.

nv = SQRT (n)

Returns the square root of ***n***. Equivalent to ***n***^.5. Causes an 'AE error' (Arithmetic error) if ***n***<0.

SQRT(N) produces the same result as N^.5, but executes more quickly.

FILE FUNCTIONS

nv = EXIST (s)

Returns a boolean TRUE if filename ***(s)*** exists, otherwise a FALSE.

nv = FADR (0)

Returns the current address of the start of the current file.

Returns 0 if there is no file opened.

nv = FGET (offset)

Returns the byte at the given offset position. The byte at offset 0 is the first data byte.

nv = FLEN (0)

Returns the number of data bytes in the current file.

nv = FREC (0)

Returns the total number of records in the file.

INTEGER FUNCTIONS

Integer functions are the same as as the normal functions, but they do their operation(s) on the integer parameters, and they return an integer as function result. Thus, execution time is considerably faster.

nv = ABS% (n%)

Returns the absolute integer value of ***n***.

nv = BAND (m%,n%)

Returns the bitwise 'AND' of the integers ***m%*** and ***n%*** .

nv = BOR (m%,n%)

Returns the bitwise 'OR' of the integers ***m%*** and ***n%***.

nv = BXOR (m%,n%)

Returns the bitwise 'XOR' (exclusive OR) of the integers ***m%*** and ***n%***.

nv = MAX% (m%,n%)

Returns the integer maximum (larger) of ***m%*** and ***n%*** .

nv = MIN% (m%,n%)

Returns the integer minimum (smaller) of ***m%*** and ***n%***.

nv = MOD% (m%,n%)

Returns the integer modulus of ***m%*** and ***n%***. Positive and negative values of ***m%*** and ***n%*** are handled as follows:

m%	**n%**	**MOD%(m%,n%)**
8	3	2
–8	3	1
8	–3	–2
–8	–3	–1

Note that this is different from the MOD function.

STRING FUNCTIONS

In the following list of string functions, the actual parameters are checked to be within the limits of the current string. Parameters that are not within limits are set to be identical to the (nearest) limit.

nv = ASC (s)

Returns the number that represents the character ***s*** (the first character of ***s***, if ***s*** is more than one character long) in ASCII notation. For example, ASC ("G") returns 71; ASC ("!") returns 33.

If ***s*** is the null string, ASC causes an 'IQ error'.

sv = CHAR (s,n)

Returns the number that represents the character at the ***n***-th position in the string ***s***, in ASCII notation.

sv = CHR$(n%)

Returns the character that is represented by the number ***n*** in ASCII notation. For example, CHR$(71) is 'G'; CHR$(33) is '!'.

sv = DATE$

Returns the string contained in the variable DATE$, representing the date. The format is

ddd mmm nn yyyy

where ***ddd*** is the day of the week, ***mmm*** is the month, ***nn*** is the date, and ***yyyy*** is the year. For example;

```
WED MAY 19 1982
```

sv = ERASE$ (s,n1,n2)

Returns a string value consisting of the characters of ***s*** from which ***n2*** characters have been deleted starting at character position ***n1***.

sv = INSERT$ (s1,s2,n)

Returns a string value consisting of the characters of ***s1*** to which the characters of ***s2*** have been inserted starting at character position ***n***.

sv = LEFT$(s,n)

Returns a string value consisting of the first ***n*** characters of ***s***.

If ***n*** is not an integer, LEFT$ truncates it.

If ***n*** is zero, LEFT$ returns the null string.

If ***n***>LEN (***s***), LEFT$ returns ***s***.

If ***n***<0 , LEFT$ returns the left part of ***s***, ***except*** for ***n*** characters (on the right of ***s***).

nv = LEN (s)

Returns the length, in characters, of the string ***s***.

sv = MID$(s,n1,n2)

Returns a substring of ***s*** beginning at the ***n1***th character, ***n2*** characters long.

If ***n1*** is 1, the substring begins at the first position in ***s***; if ***n1*** is 2, the substring begins at the second position, and so on.

If ***n1*** is not an integer value, MID$ truncates it. If ***n1***>=LEN(***s***), MID$ returns the null string.

If ***n2***=0, MID$ returns the null string. If ***n2*** is greater than the number of characters remaining in the string from ***n1*** to the end, MID$ returns the entire part of the string from character ***n1*** to the end.

If ***n2***<0, MID$ returns the ***s except*** the middle part, starting at ***n1*** with length ***n2***.

sv = RIGHT$(s,n)

Returns a string value consisting of the last ***n*** characters of ***s***.

RIGHT$ treats unusual values of ***n*** the same way that LEFT$ does.

If ***n***<0 , RIGHT$ returns the right part of ***s except*** for ***n*** characters (on the left of ***s***).

nv = SEARCH (s1,s2,n)

Searches for the ***n-th*** occurrence of the string ***s2*** in the ***s1***. When found, returns the character position of the first matching character. Otherwise returns 0 (not found).

sv = SPC$ (n%)

Returns a string of (integer) ***n*** spaces. Identical to "bbbbbbb" where the string of b's denotes ***n*** spaces.

sv = STR$(n)

Returns the value of ***n***, converted to a string. For example, STR$(71) is '71'.

STR$ uses the same conversion rules that the PRINT statement uses when it displays a numeric value.

sv = STRF$(n,width,low-exp,high-exp,power-increment, rounding-factor,frac-len,exp-len)

Returns the value of ***n***, converted into a string according to the remaining parameters. The parameters are defined as follows:

width is the total length of the resultant string. See the notes below for how ***width*** interacts with ***frac-len*** and ***exp-len***.

low-exp and ***high-exp*** are integers that determine what numbers will be formatted in floating point notation, and which will be formatted in scientific notation. Specifically, if

$n < 10^{\text{low-exp}}$

or

$n > 10^{\text{high-exp}}$

then the result will be formatted in scientific (exponential) notation; otherwise, the result will be formatted in floating point notation. (Floating point notation is the normal way a number is printed in SnapBASIC, e.g. "345.768149034").

power-increment is a value that allows you to specify in what increments the exponent will be increased when a number is formatted in scientific notation. Normally, the increment is one; a number entered as 1.45E10 would be printed as 1.45E10, just as it was entered. However, if ***power-increment*** is 3 (for example), the exponents would only be multiples of 3, so that 145E10 would be formatted as 14.5E9. This is useful if you want your numbers formatted as millions, billions, and so on.

rounding-factor is an integer that specifies at what digit rounding is to happen, in terms of powers of 10. In other words, if ***rounding-factor*** is ***x***, then all digits following the 10^x position will become 0, and the digit in the 10^x position will be rounded according to the original remaining digits. The position immediately to the left of the decimal is thus the "0th" position.

Example: Assume that ***n*** = 12345.6789. Then:

Rounding factor	***Result***
2	12300.0000
-2	12345.6800
4	10000.0000
-99	12345.6789
99	U.0000*

*a warning that nothing sensible could be printed for this value, since the rounding-factor is so high.

frac-len is the number of fractional digits (those following the decimal point) that are to included. The decimal point is ***not*** included in this count. Digits beyond the number specified by ***frac-len*** are truncated.

exp-len is the number of exponent digits to be included. The 'E' ***is*** included in this count.

The several field width parameters interact as follows, to determine the number of digits reserved for the integer part of the number:

Assume ***width*** is 16, ***frac-len*** is 5, and ***exp-len*** is 3. Then the resultant field will look like this:

```
16 positions total width
----------------
iiiiiii.fffffEee
```

The exponent, on the right, takes up three positions (including the 'E'). The fractional part is exactly 5 digits wide. The decimal point uses a position, leaving 7 digits for the integer part. If the number is negative, then there is one less position available for the integer part.

Some of the parameters have limited ranges, and some values for some of the parameters have special meanings. Further, each parameter has a "default" value that is that parameter's equivalent in the STR$ function, and in normal free format output from PRINT statements.

Parameter	**Comment**
width	Must be > 0, or causes an error. Default is 22.
low-exp	No limitation. Default: -2.
high-exp	No limitation. Default: +10.
power-increment	If < 1, converted to 1. Default: 1.
rounding-factor	No limitation. Default: -12.
frac-len	Negative values changed to 0. Special case: If ***frac-len*** = 0, as many digits as are possible will be printed following the decimal point, with trailing zeroes suppressed. Further, decimal point alignment is disabled. Default: 0.
exp-len	Values less than 0 are changed to 0. If the value is 0, decimal point alignment is disabled, and the width of the exponent field will vary according to the size of the exponent. A value of 1 will always generate a field of asterisks if there is any exponent, since the E itself requires a position. Default: 0.

This function is designed so that, upon output, decimal points will be aligned in a column (unless this is defeated by 0 values for ***frac-len*** or ***exp-len***). There are some special conditions:

- If the number is too large to represent in the space provided by the ***width*** parameter, then all that will be printed is a field full of asterisks (for example, if ***width*** is 3, then 12345 will be printed as '***'.)
- As indicated above, if ***n*** $< 10^X$, then a rounding factor X or greater will cause the integer part of the number to be replaced by a 'U'.
- If the width is insufficient to represent all the desired digits in the fractional part of the number, then the fractional part will be truncated.
- If the actual exponent cannot be represented in ***exp-len*** characters, the field is filled with asterisks.

sv = SUB$ (s1,s2,n)

Returns a string value consisting of the characters of ***s1*** from which the ***n***-th occurrence of the string ***s2*** has been deleted.

sv = TIME$

Returns the string contained in the variable TIME$, representing the time, in the format

!m!m!m hh:mm:ss

where ***hh*** is the hours (of a 24-hour clock), ***mm*** is the minutes, and ***ss*** is the seconds, e.g. "21:48:59".

nv = VAL (s)

Returns the value of ***s***, converted to a number. For example, VAL ("71") is 71.

VAL uses the same conversion rules that the INPUT statement uses when it reads a numeric value. If the string value cannot be interpreted as a number, VAL ignores everything from the first invalid character to the end of the string. Thus VAL ("5X") returns the value 5; VAL ("FGHRTY") returns the value 0.

Note that VAL, like INPUT, considers a lower case 'e' to be a valid character in a number expressed in scientific notation. For example, '2.5E3' and '2.5e3' are valid, and will return the value 2500.

TRIGONOMETRIC FUNCTIONS

SnapBASIC on the HHC has the following trigonometric functions.

nv = **COS (n)**
Returns the cosine of the angle ***n*** (radians).

nv = **PI**
Returns the value of the constant ***pi*** (3.14159265359).

nv = **SIN (n)**
Returns the sine of the angle ***n*** (radians).

nv = **TAN (n)**
Returns the tangent of the angle ***n*** (radians).

CHAPTER 3: OPERATORS

NUMERIC OPERATORS

Operators are listed in descending order of priority, ***i.e.***, the operators with the smallest "priority" numbers have the highest priority.

Numeric Operators

oper-ator	**pri-ority**	**function**	**example of use**	**result**
( )	0	overrides other priority rules	A=5*(4+3)	A=35
-	1	negation	A=-8^2	A=64
^	2	exponentiation	A=2^3	A=8
*	3	multiplication	A=2*6	A=12
/	3	division	A=8/2	A=4
+	4	addition	A=8+1	A=9
-	4	subtraction	A=8-1	A=7
=	5	is equal to	IF 5=0 GOTO 90	no action
<>	5	is not equal to	FL=5<>0	FL=1
<	5	is less than		
<=	5	is less than or equal to		
>	5	is greater than		
>=	5	is greater than or equal to		

STRING OPERATORS

String Operators

operator	**priority**	**function**	**example of use**	**result**
( )	0	overrides other priority rules		
+	4	concatenation	A$ = "8" + "1"	A$ = "81"
-	5	deletion	A$ = "abc"-"ab"	A$ = "c"
=	5	is equal to	IF "XX" = "X" GOTO 90	no action
<>	5	is not equal to	F? = "XX"<>"X"	F? = TRUE
<	5	is less than		
<=	5	is less than or equal to		
>	5	is greater than		
>=	5	is greater than or equal to		

Strings are evaluated in ASCII order. Upper case letters come before (are lesser than) lower case letters. If this does not distinguish between the strings, then the length counts: the shorter string is "smaller".

BOOLEAN OPERATORS

For these examples, assume F? = FALSE and T? = TRUE.

Boolean Operators

operator	**priority**	**function**	**example of use**	**result**
( )	0	overrides other priority rules	A? = (F? OR F?) AND (F? OR T?)	A? = F?
NOT	1	complement	A? = NOT F?	A? = T?
=	5	equivalance	A? = (F? = T?)	A? = F?
<>	5	nonequivalence	A? = (F? <> T?)	A? = T?
AND	6	conjunction	A? = F? AND T?	A? = F?
OR	7	disjunction	A? = F? OR T?	A? = T?
XOR	7	exjunction	A? = T? XOR F?	A? = T?

Truth tables for the Boolean functions are:

A?	NOT A?
T?	F?
F?	T?

A?	B?	A? = B?
T?	T?	T?
T?	F?	F?
F?	T?	F?
F?	F?	T?

A?	B?	A?<>B?
T?	T?	F?
T?	F?	T?
F?	T?	T?
F?	F?	T?

A?	B?	A? AND B?
T?	T?	T?
T?	F?	F?
F?	T?	F?
F?	F?	F?

A?	B?	A? OR B?
T?	T?	T?
T?	F?	T?
F?	T?	T?
F?	F?	F?

A?	B?	A? XOR B?
T?	T?	F?
T?	F?	T?
F?	T?	T?
F?	F?	F?

Where SnapBASIC must disregard the order of precedence to avoid a TM error (Type Mismatch), it does so automatically. Here is an example of such a situation:

```
A=A+X$=Y$
```

The normal order of precedence would evaluate 'A + X$' first. That would produce a type mismatch, however, so SnapBASIC evaluates 'X$ = Y$' first, yielding a 0 or 1, which may be added to A.

The conversions are performed according to the following table.

	INT	REAL	BOOL	STRING
INT:	INT	REAL	INT	ILLEGAL
REAL:	REAL	REAL	REAL	ILLEGAL
BOOL:	INT	REAL	BOOL	ILLEGAL
STRING:	ILLEGAL	ILLEGAL	ILLEGAL	STRING

In short, operators are coerced to the type that requires the most storage: BOOLEANs are shorter than INTs, which are shorter than REALs. STRINGs are not coercable, and nothing is coercable to STRING.

Certain numeric operations are converted to logical operations, when the operands are Boolean. Specifically:

+ is converted to OR
* is converted to AND
- is converted to NOT

Also, NOT is converted to - when used with numeric operators.

Certain other operations are simply illegal for certain types. The operations *, /, and ^ are illegal for STRING operands. The operations -, /, and ^ are illegal for BOOL operands. Using these operations of illegal types will cause the TM error.

CHAPTER 4: RESERVED WORDS

ABS(	FOPEN	PAREN
ABS%(	FOR	PEEK(
AND	FPUT	PI
ASC(	FREAD	POKE
ATTACH	FREC(	POS(
AUTO	FREE(	PRINT
BAND(	FREVISE	?
BOR(	FWRITE	READ
BURN	GET	REM
BXOR(	GOSUB	RESEQ
BYE	GOTO	RESTORE
CALL	HISTORY	RETURN
CHAR(	HOME	RIGHT$(
CHR$(	IF	RND(
CLEAR	INPUT	ROUND(
CONT	INSERT$(	RUN
COS(	INT(	SEARCH(
DATA	LEFT$(	SIN(
DATE$	LEN(	SQR(
DEF	LET	SQRT(
DEL	LIST	SQUEAK
DETACH	LN(	SPC$(
DIM	LOAD	STEP
END	LOG(	STOP
ERASE$(	MAX(	STR$(
EXIST(	MAX%(	STRF$(
EXP(	MID$(	SUB$(
FADR(	MIN(	TAN(
FALSE	MIN%(	THEN
FDEL	MOD(	TIME$
FGET(	MOD%(	TO
FINS	NEW	TROFF
FIX(	NEXT	TRON
FLEN(	NOT	TRUE
FLOAT(	ON	VAL(
FNx(	ONERR	XOR
	OR	

CHAPTER 5: LINE NUMBERS, NUMBERS, AND STRINGS

LINES

Maximum length: 80 characters (after SnapBASIC has LISTed a line on the display.)

Minimum value: 0

Maximum value: 32766

Values allowed: any integer value from minimum to maximum.

NUMERIC VALUES

Maximum value (magnitude): approximately $\pm 9.99999*10^{1023}$

Tiniest non-zero value: approximately $\pm 1.00000*10^{-1024}$

Maximum precision: 13 decimal digits

Memory used: 8 bytes per value

INTEGER VALUES

The limits on integer values apply only to the values of integer ***variables*** and ***arrays***.

Maximum magnitude: 32767

Minimum value: -32767

Memory used: 2 bytes per value

BOOLEAN VALUES

The limits on boolean values apply only to the values of boolean ***variables*** and ***arrays***. A boolean variable may have an "integer value" (a value that contains no decimal part) 0 and 1 .

Only possible values: TRUE and FALSE

Memory used: 1 byte per value

STRING VALUES

A string value contained in a string constant, variable, or array, contains ASCII characters represented in 7 bits.

String Length: 0 to 255 bytes, inclusive

Memory used: 3 bytes plus characters in the string, allocated dynamically.

ARRAYS

Maximum number of dimensions: 13 (more causes CX error)

Maximum number of elements per dimension: as many as can fit in memory; absolute maximum of 32766

Memory used: 6 + 2 * (number of dimensions) + (total number of elements) * (size of each element)

RULES FOR DISPLAYING REAL VALUES (PRINT, STR$, AND STRF$)

1. If a number is negative, the first character displayed is '-'. Otherwise the first character displayed is "space".

   The number follows. It contains only as many digits as are needed to represent it completely. There are no leading zeroes before the decimal point (if any), and no trailing zeroes after the decimal point (if any).

   The number is not divided by commas into groups of three characters, although such commas are used in some parts of this book to make large numbers more readable.
2. If the absolute value of the number is in the range

   .01 <= ***n*** < 1,000,000,000

   the number is displayed as a real value (with a decimal point). ***Examples:***

   ```
   .1     -.1     3.141592     9111.11111
                 -9111.11111
   ```
3. If the number doesn't fit into category (2) or (3) above, it is displayed in scientific notation.

   The mantissa is expressed as a number in the range 1<=***n***<10 displayed as a real number. ***Example:***

   3.14E+14

   When 'E' is negative, it is followed by a '-' and an exponent of maximum 4 digits. ***Examples:***

   3.14E14   3.14E-14   3.E-3

RULE FOR DISPLAYING INTEGER VALUES (PRINT)

If the number is negative, it is preceded by a minus sign. The number follows, without a trailing decimal point. There are no extra spaces.

RULE FOR DISPLAYING BOOLEAN VALUES (PRINT)

Boolean values are displayed as TRUE or FALSE.

Note that TRUE has a trailing space, so that both are 5 characters long.

RULE FOR DISPLAYING STRING VALUES (PRINT)

Strings are displayed as entered, without extra spaces.

RULES FOR READING REAL VALUES (INPUT AND VAL)

The rules for reading numeric values are essentially the same as the rules for displaying them, except that the rules that choose between alternate forms of a number do not apply. For example, any number, regardless of value, may be input in scientific notation, or as a decimal value, or (value permitting) as a integer.

1. If a number is negative, the sign must be '-'. Otherwise the sign may be '+' or may be absent. Leading blanks are allowed.
2. After the sign, if any, is an integer constant or numeric constant.

   SnapBASIC considers the number to stop at the first character that is not part of a valid value. For example, if the value being read is '50,000', SnapBASIC reads '50'. If the number is ' 50' (with a leading blank), SnapBASIC reads '50', since leading blanks are allowed.

   Note that a leading zero is not required; the numbers '0.1' and '.1' are equivalent.

If the number is in floating point notation, it ends here. If the number is in scientific notation, the following parts must be present.

3. The next character is 'E' (it may be in upper or lower case).
4. Next is the exponent's sign. If the exponent is negative, its sign is '-'. Otherwise its sign may be '+', or may be absent.

5. Next is the exponent, which must be an integer.
6. The entire number must be within the valid range of a numeric variable.

   For example, all of the following numbers are valid and equivalent:

3.14E+14	3.14E14	3.14E+0000014
314E+12	314000E9	+0.314E+15
.314E15		

There are a few special cases. A decimal point entered by itself results in the number 0.0; a decimal point followed by an E followed by an exponent results in the number 1.0Eexp; the same applies to a 0 followed by an E followed by an exponent. If an E is used at all, it ***must*** be followed by an exponent or an error will occur.

RULES FOR READING INTEGER VALUES (INPUT AND VAL)

The rules for reading integer values are the same as the rules for reading real values; once the value is entered, SnapBASIC attempts to convert it to an integer. If the number is not within the range of an integer, an IQ error results.

RULES FOR READING STRING VALUES (INPUT AND VAL)

Any line of text constitutes a string. Strings are terminated by a comma. If it is necessary to enter a comma as part of a string, the entire string must be enclosed in quotes ("). If it is necessary to include a quote as part of a string, two quotes must be used and the string must be enclosed in quotes. Examples:

```
this is one single string
this is two strings, since there is a comma
"but this, with commas and quotes, is one string"
"this string contains """quotes""."
```

RULES FOR READING BOOLEAN VALUES (INPUT AND VAL)

Boolean values may be entered as arbitrary strings. Any string starting with upper or lower case 'N' or 'F', or starting with the digit 0, is interpreted as FALSE; any other string is interpreted as TRUE.

CHAPTER 6: BASIC PROGRAM EDITOR QUICK REFERENCE

EDITING LINES

Change to be made	**Procedure**
Add a line	Type a line with a line number before it. Basic inserts the line in the program in line number order.
Delete a line	(1) Type a line number alone; press ENTER.
	or
	(2) EDIT the line (by using the arrow keys) and delete everything except the line number, then press ENTER to make Basic accept the edited line.
	or
	(3) "DELETE linenr" to DELETE one line
	or
	(4) "DELETE linenr1,linenr2" to DELETE lines from linenr1 to linenr2, inclusive.
Change a line	Type a new line with the same line number as the line you want to replace. You can use all the usual HHC line edit commands.
	or
	LIST the line and edit it.
List a line [1]	(1) 'LIST' will list the whole program
	or

[1] - If a LISTed line appears in inverse characters, the sprcitM 4 S version of the line expanded by LIST is longer than 80 characters, and information is not available in the display although the compiled line may be RUNable. If this line must be changed, the missing information must be added back and the line shortened.

	(2) 'LIST linenr' lists the line and goes to edit mode ***or*** (3) 'LIST linenr, linenr' lists the range of line numbers To stop a LIST, press the BREAK key.
Copy a line	EDIT the line and change its line number to the line number you want the copy to have. Press ENTER to make Basic accept the edited line.
Move a line	Copy the line; then delete the original line.

ARROW KEYS

Key	**Function**
Up arrow	If not already in EDIT mode, enter EDIT mode at the "current" line (this is the last line EDITed if there was one; otherwise, it is the last line in the program). If already in EDIT mode, updates the current line, and moves to the previous line. If this was the first line in the program, return to command mode. In this case, another up arrow will go to the last line.
Down arrow	If not already in EDIT mode, behaves the same as up arrow: except that if there was no last line EDITed, the "current" line is the last line in the program. If already in EDIT mode, updates the current line, and moves to the next one, if there is one; otherwise returns to command mode. In this case, another down arrow will go to the first line.
Right arrow	Moves cursor right. To move to the end of line, hold down the key until the HHC beeps twice (signaling end of line).
Left arrow	Moves cursor left. To move to the beginning of the line, hold down the key until the HHC beeps twice (signaling beginning of line).

INSERT CHARACTERS

Editing Operation	**Keystrokes**
Insert a character, ***x***	INSERT ***x***
Insert multiple characters	LOCK INSERT ***xxxxx...*** Press INSERT again when done.
Insert a character to right of cursor	INSERT right arrow key, new key
Insert a character to left of cursor	INSERT left arrow key, new key

DELETE CHARACTERS

Editing Operation	**Keystrokes**
Delete character under cursor; leave cursor on following character	DELETE right arrow
Delete character under cursor; move cursor left to preceding character	DELETE left arrow
Delete to end of line	LOCK DELETE right arrow (hold down arrow)

SHORTCUTS

Editing Operation	**Keystrokes**
Review a line longer than LCD	ROTATE Press right arrow key twice when done.

Go directly from any INSERT mode to any DELETE mode	Skip pressing INSERT. Press DELETE, LOCK DELETE, etc.
Go directly from any DELETE mode to any INSERT mode	Skip pressing DELETE. Press INSERT, LOCK INSERT, etc.
Interrupt a "LOCK" operation before its natural end	Press any key.

Note: If you are in the middle of editing any line, you may break without updating (i.e., leaving the compiled line unchanged) by pressing the BREAK key.

CHAPTER 7: PERIPHERAL DEVICES

CONTROL CHARACTERS

Control characters are "characters" which do not generate output on an output device, but cause the device to perform control functions, such as cursor movement.

There are three categories of control characters for you to be concerned with:

1. ASCII standard control characters which are recognized by the HHC. These characters are listed in the following table with numeric values below 32.
2. ASCII standard control characters which are not recognized by the HHC. These are all the characters represented by numeric codes below 32 that are ***not*** listed in the following table.
3. HHC control characters that are not standard ASCII control characters. These are characters listed in the following table with numeric values above 32.

If you write a control character to a device that does not use it, the device will either ignore the character or display its HHC graphic representation. See the descriptions of individual devices for details.

ASCII Control Characters

numeric value	**name**	**ASCII std**	**typical meaning**
7	bell	yes	Sounds audible alarm.
8	backspace	yes	Move cursor left one position; erase character cursor is moved to.
10	line feed	yes	Move cursor down one line.
12	form feed	yes	Move cursor to start of first line on next page.
13	carriage rtn	yes	Move cursor to start of next line. (Note that "carriage return" does an automatic line feed on HHC peripherals.

			This differs from its action on some other devices.)
27	escape	yes	Marks beginning of an escape control sequence.

ESCAPE CONTROL SEQUENCES

Escape control sequences provide an extended set of control characters on the HHC. Each sequence is 3 bytes long:

byte	**meaning**
0	escape (ASCII value 27) begins an escape control sequence.
1	operation code (opcode).
2	data; meaning depends on the opcode. Unless noted otherwise below, the data byte is ignored.

The HHC has a standard set of escape control sequences which apply to all peripherals. Not all peripherals respond to all escape control sequences, however. If a peripheral does not respond to a particular escape control sequence, it ignores that sequence; that is, the sequence is a "no-operation" command for that peripheral.

In some other HHC manuals the opcodes are referred to by five-character symbols. These symbols are included in the following table.

Opcodes

Name: LCD Unescape

Opcode: 64

Symbol: ESCUN

On the LCD, the data byte is displayed (as in ESCDA). On all other devices, it is ignored.

Name: Insert Right

Opcode: 65

Symbol: ESCIR

The data byte is displayed at the cursor. The character previously under the cursor, and all following characters on the line, are pushed to the right.

Name: Delete Right

Opcode: 66

Symbol: ESCDR

The character under the cursor is deleted; following characters on the line are moved left. The data byte is used as a fill character at the end of the line.

Name: Set Inverse Mode

Opcode: 67

Symbol: ESCSI

Subsequent output is displayed in inverse-image (the colors of the character and background are reversed).

Name: Set Uninverse Mode

Opcode: 68

Symbol: ESCUI

Subsequent output is displayed normally (not inverse-image). Reverses the effect of ESCSI.

Name: Set Flash Mode

Opcode: 69

Symbol: ESCSF

Subsequent output will be displayed in flashing characters.

Note: the advent of flashing may be delayed (***i.e.***, characters written immediately after "set flash mode" may not flash) under some circumstances. You can force flashing to begin by doing an I/O operation on any device other than the LCD.

Name: Set Unflash Mode

Opcode: 70

Symbol: ESCUN

All subsequent output is displayed in non-flashing characters, until the mode is changed by ESCSF. Reverses the effect of ESCSF.

Note: the end of flashing may be delayed, like the advent of flashing (see above).

Name: Display Character Absolute

Opcode: 71

Symbol: ESCDC

The data character is displayed, even if it is a control character that would normally be executed. For example, if the next data character is 13 (carriage return) and it is sent to the micro printer, it is written as an inverse-image M.

Name: Flush I/O Buffer

Opcode: 72

Symbol: ESCFL

Characters in the device's I/O buffer are written (if they are being output) or discarded (if they are being input), emptying the buffer.

This operation is generally applied only to output devices that write a line of data at a time. Writing a line would normally be triggered by a CR character.

Name: Set Control Character Mode

Opcode: 73

Symbol: ESCCC

If the data byte is non-zero, subsequent non-executable control characters sent to the device will be displayed; if zero, subsequent non-executable control characters sent to the device will not be displayed.

Name: Home Cursor

Opcode: 74

Symbol: ESCHM

Returns the cursor to the upper left corner of the display.

Name: Set Word Break

Opcode: 75

Symbol: ESCWB

The data byte defines the **word break character**. When the device encounters this character in output data, it considers the character to mark the break between two words.

When an output line becomes longer than the device's maximum line length, the device **word-wraps** automatically; that is, it ends the line and moves the last word of the line down to the next line, so that the word will not straddle a line break.

The initial value of the word break character is 32 (space) for each device.

If you set a device's word break character to 255, word wrapping is disabled for that device.

Note that the LCD has no word-wrap capability.

THE KEYBOARD

The keyboard is normally attached to LUN #0. See Chapter 9 for the displayable characters.

Technical Information

ATTACH device code: 129

Control codes: not applicable to input-only devices.

Escape control sequences: not applicable to input-only devices.

THE LCD

The LCD is normally attached to LUN #1. See Chapter 9 for the displayable characters.

Technical Information

ATTACH device code: 65

Control codes:

Code	Meaning	Function on device
7	Bell	Makes the HHC beep.
8	Backspace	Backspaces the cursor, erasing the character the cursor was previously at. [1]
12	Form feed	Prints an inverse upper case L.
13	Carriage rtn	Clears display and moves cursor to left edge of LCD.
27	Escape	Begins an escape control sequence.

Escape control sequences: the LCD supports all of the standard escape control sequences except ESCFL, ESCCC, and ESCWB. Note that the data byte of ESCUN is displayed on the LCD (as in ESCDC); it is ignored on all other devices.

THE TV ADAPTOR

The TV adaptor may be coupled to a standard black-and-white or color television receiver or video monitor. It provides a two- dimensional display that is more useful for many purposes than the one-line display on the HHC's LCD.

The TV Adaptor display shows 16 lines, each 32 characters long. It can also generate several kinds of dot-matrix graphic displays. On a color television set, it can display letters, graphics, and background in various colors.

[1] - if the cursor is in the leftmost character position, the backspace will shift all existing characters to the right and insert a blank space in the first character position.

Technical Information

ATTACH device code: 67 (output)

Control codes and escape control sequences:

The TV Adaptor has many control codes and escape control sequences, and a description of them would be much too long to include in this manual. Information on programming for the TV Adaptor may be found in a special publication.

THE MICRO PRINTER

The micro printer prints 15-column lines on a roll of paper 1.4' wide. It has a "thermal" printing mechanism that makes marks on specially coated paper by heating tiny areas in the print head. The character set is identical to that of the LCD.

The micro printer uses a buffer in the HHC's RAM that can hold up to two lines of data. The printer accumulates two lines at a time, and prints them in a single operation.

Technical Information

ATTACH device code: 68

Control codes:

Code	Meaning	Function on device
7	Bell	No-operation.
8	Backspace	"Erases" the last character sent to the printer, if it has not already been printed. Several backspaces in a row will erase several characters.
10	Line feed	No-operation. The micro printer automatically advances the paper when it does a carriage return. A separate line feed operation is not supported.
12	Form feed	Ends the current line of output, prints the contents of the printer's buffer, and advances the paper 4 lines.
13	Carriage rtn	Ends and prints the current line of output. If the buffer contains two lines of output, it prints both.
27	Escape	Begins an escape control sequence.
81	Cursor left	Non-destructive backspace.
82	Cursor right	Non-destructive space.

Escape control sequences: ESCCC, ESCDC, ESCDR, ESCFL, ESCHM, ESCIR, ESCUN, and ESCWB.

SERIAL INTERFACE

The serial interface enables the HHC to do I/O on a great variety of devices designed to communicate through an **RS232C interface**. This interface, established by the Electronics Industries Association (EIA), is widely used for low- and medium- speed peripheral devices.

Initializing the Serial Interface

Before you use the serial interface for the first time, you may need to use the RS232C capsule's configure option to make the interface compatible with the device you want it to control. The configuration program sets properties such as the type of error checking the serial interface is to do.

To configure the serial interface, plug the interface into the HHC and turn it on with the I/O menu; select the "Serial I/O" program from the primary menu, and then select the "Configure" option from the program's menu. (Note that this is done from the HHC Main Menu, rather than from BASIC. It is possible to initialize the serial interface from a BASIC program, but the process requires some understanding of the HHC's machine language, and is beyond the scope of this manual.)

The configuration program creates an "invisible" file which contains initialization data. {2} Whenever you use the serial interface, the HHC automatically reads this file and initializes the serial interface from the information contained in it. Thus, you need not run the RS232C configure program again unless the initialization file is somehow deleted.

The initialization file can contain a separate set of data for the bus socket on the HHC (slot #0 in the I/O key menu) and for each socket in the I/O adaptor (slots #1 through #6 in the menu). Thus, you can set up the initialization file so that you can change the interface's configuration just by plugging it into a different slot.

If you have some computer experience, you will probably find the configuration program to be self-explanatory. If you do not,

{2} - an invisible file is one that does not appear in the file system editor's menu or in BASIC's menu.

see the manual that accompanies the RS232C capsule for instructions.

Other capsules such as Telecomputing 1 and 2 will also operate in the Serial Interface, offering additional interface possibilities.

Note About Protocols

The RS232C capsule's configuration program can make the serial interface operate with or without a **transmission protocol**. This is a set of rules that a computer and a peripheral (or two computers) can use to make sure that neither one sends characters when the other is unable to receive them.

Whether or not you initialize the serial interface for a transmission protocol must depend on whether the interface is connected to a device that uses one. Unless the serial interface and the device connected to it are using the same protocol, they cannot communicate properly.

The serial interface supports two alternate software protocols: XON/XOFF protocol and ETX/ACK protocol. Additionally, the Data Terminal Ready (DTR) line in the data transmission cable controls communication.

XON/XOFF protocol is commonly used in communications between the HHC and another computer. Many kinds of printers and other peripheral devices use it as well.

Suppose you are using XON/XOFF protocol to communicate between the serial interface adaptor and a printer. Here is how the protocol works.

As the printer receives characters from the serial interface, it stores them in a buffer until it can print them. If the serial interface sends characters to the printer faster than the printer can print them, the printer's buffer eventually fills up. When the buffer is almost full, the printer sends the interface an **XOFF** command ("transmission off", ASCII code #19). This makes the interface stop transmitting. When the printer's buffer becomes less full, the printer sends an **XON** command ("transmission on", ASCII code #17). This makes the interface resume transmitting.

XON/XOFF protocol works for input to the HHC, as well as output from it. Suppose you were using the serial interface to communicate with another computer, which could both send and receive characters. If the other computer sends characters faster than the HHC can process them, eventually the serial interface's buffer nearly fills up. Then the serial interface sends an XOFF command to make the other computer stop transmitting. Later the serial interface sends an

XON command to make the other computer resume transmitting.

XON/XOFF protocol has two effects on you as a user of the serial interface:

1. You do not have to worry that the device attached to the interface might lose data because the HHC continued transmitting when the device's buffer was full. The serial interface handles the XON/XOFF protocol automatically, and prevents any such mishap from occurring.
2. You cannot transmit or receive the ASCII codes #17 (XON) and #19 (XOFF) as data characters (except as part of an escape control sequence). If you try, the device connected to the serial interface will interpret the characters as XON and XOFF commands. That will make it start or stop transmitting data at inappropriate times.

ETX/ACK protocol is commonly used by peripheral devices such as printers. Its purpose is the same as the purpose of XON/XOFF protocol: to make sure that the serial interface does not send information when the attached device is unable to receive it.

In ETX/ACK protocol, the serial interface transmits a string of characters, called a **message**, that is known to be short enough for the device to process without losing characters. The interface ends the message with an ETX character ("end transmission", ASCII code #3). When the device has processed the message and is ready for another one, it sends an ACK character ("acknowledge", ASCII code #6). This signals the serial interface that it may send another message.

Unlike XON/XOFF, ETX/ACK protocol works only for transmissions in one direction: from the Serial Interface Adaptor to a device. Thus it is unsuitable for devices that engage in two-way communication, such as modems and keyboard printers.

The effects of ETX/ACK protocol on you as a user are the same as the effects of XON/XOFF protocol, except that the ASCII character you cannot transmit as data (except as part of an escape control sequence) is #3 (ETX) instead of #17 (XON) and #19 (XOFF). (You can transmit ACK as a data character, since it has a special meaning only when ***received*** by the Serial Interface.)

Technical Information

ATTACH device codes: 70 (output) and 134 (input)

Control codes: none; ***but see*** the discussion of transmission protocols, above.

Escape control sequences: none. The "escape" control character (ASCII code 27) is treated as data.

THE MODEM

The modem enables you to communicate with other computers via telephone. Two rubber cups on the modem hold the mouthpiece and earpiece of a standard telephone handset.

The modem encodes the information that the HHC writes to it in sound patterns and transmits them over the telephone. It receives information in the same fashion, and passes it to the HHC when the HHC "reads" the modem.

The modem contains an object called a **control ROM**, which is similar to an HHC capsule, but contains a program to control the operation of the modem itself, as well as an application program that you can run from the primary menu.

The modem's control ROM is interchangeable in much the same way that an HHC capsule is. Two control ROMs are available for the modem at this time; their names are Telecomputing 1 and Telecomputing 2. Their functions are similar, but Telecomputing 2 has more features than Telecomputing 1 does. For details on the features of these programs, study the instructions that accompany the modem, and speak to the distributor of your HHC.

Initializing the Modem

Before you use the modem for the first time, you may need to use the telecomputing system's configuration selection to make the modem compatible with the computer you want the modem to communicate with. The procedure for configuring the modem is very similar to the procedure for configuring the serial interface adaptor, described above. The major differences are:

1. The modem's configuration file can hold only one set of configuration data, rather than one set per I/O slot, as the serial interface's configuration file does.
2. The modem supports XON/XOFF protocol, but does not support ETX/ACK protocol. (Telecomputing 1 sends XON/XOFF to the host computer, but does not "listen" for them. Telecomputing 2 can send and/or listen for XON/XOFF.)

Technical Information

ATTACH device code: 130 (input) and 66 (output)

Control codes: none; ***but see*** the notes on XON/XOFF protocol above, and under the description of the serial interface adaptor.

Escape control sequences: none. The "escape" control character (ASCII code 27) is treated as data.

CHAPTER 8: PEEKS AND POKES

INPUT AND OUTPUT

The System Device Table (SDT)

The HHC keeps track of LUN attachments through the **System Device Table** (**SDT**). The SDT is kept at locations 705 through 712.

The byte at:	represents LUN
705	#0 (normally the keyboard)
706	#1 (normally the LCD)
707	#2
708	#3
709	#4
710	#5
711	#6
712	#7
713	#8
714	#9
715	#10
716	#11
717	#12
718	#13
719	#14
720	#15

Interpret the value of each byte as follows:

value	means
0	Keyboard is attached to this LUN.
6	LCD is attached to this LUN.
255	Nothing is attached to this LUN.
other	A peripheral is attached to this LUN. Values indicate the order in which peripherals were ATTACHed, not peripherals' device types.

To unattach a device, POKE 255 into the proper SDT entry. This is the same as executing the DETACH command for that LUN, except that DETACH also turns off the device.

Treat LUNs #0 and #1 ***very carefully***, since they are your channels for communicating with the HHC. If your program should leave LUN #0 without a properly attached device, you will be unable to control the HHC; if it leaves LUN #1 without a properly attached device, you will be unable to see what you are doing. Either way, you may have to press CLEAR to reset the device attachments.

THE HELP AND I/O KEYS

The I/O key is normally disabled in SnapBASIC.

To make the HELP and/or I/O keys function while you are in SnapBASIC, POKE the following values into location 524:

value	**means**
0	HELP and I/O both function
1	HELP functions (the normal case)
4	I/O functions
5	neither key functions

Note: If you have used the POKE to activate the HELP function, your setting in location 524 may change after the HELP key is actually pressed.

Warning: You can view the I/O menu from within SnapBASIC, but it is not possible to make changes.

THE KEYBOARD BUFFER

The HHC stores keystrokes that it has not yet processed in a **keyboard buffer**. You can use PEEK to look ahead at the contents of this buffer before you do an INPUT or GET, and you can use POKE to "type" into the buffer, so that your program, in effect, is pressing keys on the keyboard.

Note: you must be very careful because one character can be removed from the buffer for each BASIC instruction executed.

Structure Of the Keyboard Buffer

The keyboard buffer is 8 bytes long. The bytes in the buffer are numbered 0 to 7. The HHC places the first character typed in the 7th byte, the second in the 6th byte, etc. After the 8th character typed has been placed in the 0th byte, the 9th character typed is placed in the 7th byte (assuming the 1st character typed has been read by the program), and so on.

The HHC maintains two pointers to the keyboard buffer. A "store" pointer contains the number (0 to 7) of the byte where the next character typed on the keyboard will be stored. A "fetch" pointer contains the number of the byte where the next character read by the program will come from.

For example, suppose you have just entered BASIC and nothing has been typed yet. [1] The keyboard buffer and its pointers look like this:

```
_ _ _ _ _ _ _ _   keyboard buffer
              ↑   "store" pointer
              ↑   "fetch" pointer
```

Now suppose you type in 3 characters, 'ABC'. Your program does a GET, so that you have input 3 characters, and your program has read one. Now the buffer looks like this:

```
_ _ _ _ _ C B A   keyboard buffer
        ↑         "store" pointer
            ↑     "fetch" pointer
```

You continue typing in the alphabet, and your program continues GETting characters. At some later time when you have typed the alphabet through J and your program has read it through E, the buffer looks like this:

```
H G F E D C J I   keyboard buffer
          ↑       "store" pointer
    ↑             "fetch" pointer
```

The Pushkey Buffer

The HHC has a second buffer called a **pushkey buffer** which it uses to hold characters that are "pushed" back into the input stream by a program.

Whenever BASIC does an INPUT or GET, the HHC returns any characters that are in the pushkey buffer before going to the keyboard buffer. Thus, any characters you store in the pushkey buffer will be read ***before*** characters typed in through the keyboard, even if the keyboard characters go into their buffer first.

The pushkey buffer is 4 characters long. It is used as a LIFO queue (the last character put in is the first taken out). The "bottom" of the buffer, where the first character is pushed, is character 0; the "top", where the last character may be pushed, is character 3.

---

[1] - This is an oversimplification, since the same buffer is used by the rest of the HHC. Something had to have been typed for you to have entered SnapBASIC from the primary menu. The discussion of the process is accurate, however.

A pushkey counter indicates the number of characters already in the buffer. Its value may be 0 to 4. A value of 0 means the pushkey buffer is empty; 4 means the pushkey buffer is full, and there is no more space for characters to be pushed into it.

For example, if you push a '2', then a 'G' into the buffer, the buffer looks like this:

```
2 G _ _   pushkey buffer
          pushkey counter = 2
```

INPUT or GET will receive the 'G', then the '2':

```
2 _ _ _   pushkey buffer
          pushkey counter = 1

_ _ _ _   pushkey buffer
          pushkey counter = 0 (empty)
```

Buffer Locations

location	contains
620	keyboard buffer (location of character 0)
518	"store" pointer to keyboard buffer
519	"fetch" pointer to keyboard buffer
628	pushkey buffer (location of character 0)
522	pushkey pointer

PEEKs and POKEs

To inspect the pushkey buffer, PEEK at the pushkey pointer. If it is 0, the pushkey buffer is empty. If it is not zero, use it to extract the contents of the pushkey buffer.

The following subroutine assembles a string whose value is the current contents of the pushkey buffer:

```
1100 REM PK$ returns contents of
       pushkey buffer.
1110 REM PC is pushkey ptr.
1120 PK$="":PC=PEEK (522)
1130 REM Accumulate characters from
       buffer.
1140 PC=PC-1:IF PC^0 THEN RETURN
1150 PK$=PK$+CHR$(PEEK (628+PC))
1160 GOTO 1140
```

To POKE a character into the pushkey buffer, check to make sure it is not full. If it is not, POKE the character into the bufer position indicated by the pushkey counter, then increment the counter.

To POKE a character into the "top" of the keyboard buffer, so that it will be the next character to come out: first, check to make sure that the buffer is not full. Then move the "fetch" pointer backwards one location, and POKE the character to the location the "fetch" pointer now indicates.

Do ***not*** try to put a character into the "bottom" of the keyboard buffer by POKEing into the location indicated by the "store" pointer and advancing the pointer. If you do this, there is always a risk that you will POKE a character at the same time that the real keyboard inputs a character; if this happens, one character or the other will be lost.

FUNCTION KEYS

The definitions of the three function keys are kept in three consecutive areas, each 16 bytes long. Each area begins with a byte containing the length of a function key definition in characters, followed by 15 bytes containing the definition. If the definition is less than 15 characters long, the part of the area beyond the end of the definition is ignored.

The locations that contain the function key definitions are:

location	contents
642	Length of f1's definition.
643	f1's definition.
658	Length of f2's definition.
659	f2's definition.
674	Length of f3's definition.
675	f3's definition.

You can change the definition of a function key by POKEing appropriate values into the that function key's definition area.

"Typing" a Function Key

You can "type" a function key by POKEing it into the keyboard buffer, but the preferred way to do it is to do the two POKEs described below.

Let FA = 642, the location of the length of F1's definition; then:

to "type" a function key	perform POKEs
f1	POKE 520,PEEK (FA) POKE 521,1
f2	POKE 520,PEEK (FA + 16) POKE 521,17
f3	POKE 520,PEEK (FA + 32) POKE 521,33

When you "type" a function key in this way, your program will INPUT it before any of the characters in the keyboard buffer or the pushkey buffer.

You cannot "type" a function key by POKEing it into the pushkey buffer. If you try, your program will INPUT the ASCII code that represents the function key (#21, #22, or #23) instead of the function key's current definition.

THE STOP/SPEED KEY

The HHC's LCD rotation speed and menu speed are controlled by the value at location 535. This location may be PEEKed or POKEd. The value's meaning is:

value	STOP/SPEED setting
10	1 (slowest setting)
9	2
8	3
7	4
6	5
5	6
4	7
3	8
2	9
1	0 (fastest setting)
0	faster than fastest STOP/SPEED setting

Caution: POKEing a value of '0' also disables the keyboard auto-repeat feature that most keys have.

DATE AND TIME

The HHC maintains a the current date and time in a 5-byte memory area at locations 526 through 530. You can PEEK at this area to get the current date and time.

The HHC's timer is not directly available to you; locations 526 through 530 contain a ***copy*** of it. This has two consequences:

1. The date and time that you can PEEK are not absolutely accurate. They are updated periodically by the HHC. The updating schedule is too complicated to explain here in detail, but at a minimum, the date and time are updated whenever one of the following events takes place:

   a. The cursor flashes on (every 0.7 seconds, when the cursor is flashing).

   b. When a character is input to the HHC from the keyboard or from any peripheral. (This refers to the physical event of inputting a character, not to the program's execution of an INPUT statement, which may happen much later.)

   c. Approximately every 9 hours.
2. You cannot POKE the date and time. If you try, the value you POKE will be wiped out the next time the HHC updates the timer.

Format Of the Date and Time

The date and time are kept in a single integer number that is 5 bytes long. The value of this integer is the number of **clock units** (one clock unit = 1/256 second) from the beginning of January 1, 1980, to the present. The bytes of the date and time count the following units of time:

location	counts units of:
526	1/256 second
527	seconds
528	256 seconds
529	65,536 seconds (approximately 18 hr., 12 min.)
530	16,777,216 seconds (approximately 194 days, 4 hr.)

One reasonable way to use the date and time is to define a 5-element array and move each byte of the value into one element:

```
500 DIM TI(5)
510 FOR I=0 TO 4
520 TI(I)=PEEK (526+I)
530 NEXT I
```

Then you can write a variant of our day-of-year calculator to convert the 5-element array into a meaningful date and time.

You can simplify the task somewhat by ignoring the first byte of the time, since the time will seldom be accurate to more than a second.

If you are concerned only with elapsed time, you can build two arrays like TI, above, one for a start time and one for an end time. Then you can "subtract" one array from the other. Do this by analogy with the ordinary process of subtracting two numbers by hand; treat each element of the array as a "numeral" and borrow from the next greater element when necessary:

```
800 REM ts is start time,
     te end time, tl elapsed time.
810 FOR I=0 TO 5
820 IF TS(I)>TE(I) THEN
     TE(I)=TE(I)+256:TE(I+1)=TE(I+1)-1
830 TL(I)=TE(I)-TS(I)
840 NEXT I
845 REM ET is elapsed time in sec.
850 ET=1/256*TL(0)+TL(1)
860 ET=ET+256*TL(2)+65536*TL(3)
870 ET=ET+16777216*TL(4)
```

ROTATION MODE

The LCD's **rotation mode** is controlled by the value at location 534. The value's meaning is:

value	rotation mode
0	fill mode. The LCD is filled with text, left to right, as fast as a program can display it. After the LCD is full, the HHC erases everything and starts filling the LCD again.
1	fill-and-rotate mode. The LCD is filled with text as in fill mode. After the LCD is full, the HHC shifts characters off the left end to make room for new characters on the right. (This is the HHC's normal mode of operation.)
2	rotate mode. Characters are rotated onto the LCD from the right edge, even when the LCD is not full of text.

Note that the rotation mode is ***not*** reset by the CLEAR key.

POKES AND PEEKS FOR FILE TYPES

The location FADR(0) + 2 contains the type byte of the current file. For some files, the first data byte is used as an "extension" to the type byte. The file type assignments are:

Bit pattern:

80H	40H	20H	10H	08H	04H	02H	01H	Designation
0	0	0	0	0	0	1	X	Capsule image file
0	0	1	0	0	0	0	0	SnapBASIC program file
0	1	0	0	0	0	0	0	PORTACALC Portafile
1	1	0	0	X	X	X	X	File naming conventions are used to distinguish files
1	1	1	0	X	X	0	X	File extension used
0	0	1	0	0	0	1	0	Temporary file

X = Value may be either 0 or 1

1. The 'temporary attribute', (bit 02H) has been reassigned to expand the list of possible file types. Attributes still in effect are:

Bit	Type	Description of effect
10H	Microsoft Basic	File appears in Microsoft Basic menu
08H	Text	File can be edited using the file system
04H	Executable	File appears in 'RUN SNAP' menu
01H	Invisible	File does not appear in file system list of files

2. A code of 0C0H through 0CFH indicates that there is some information in the file NAME itself that can be used to distinguish file types. Various conventions may be used to identify file classes. For example, PORTAFLEX source files

have ".SRC" as the last four characters in their name; PORTAFLEX object files have ".OBJ" as the last four characters.

The file extension byte (to be used only with a file type byte of 0E0H) has the following re-definition:

File Extension Byte	**Usage**
0F0H - 0FFH	Further expansion bytes used
0C0H - 0EFH	User-definable extension byte
080H - 0BFH	Reserved for future use
0 - 07FH	Reserved for Friends Amis

Two extension bytes have been already allocated;

0H	PortaWriter compressed text file
10H	SnapFORTH dictionary file

Applications should check for explicit 8-bit combinations when usign the file extension byte feature. Any use of the user-definable set of extensions should be for very special purpose software in which the chance of a conflict is very small. Friends Amis will assign file type extensions to software developers who are developing general purpose software.

The attribute bit values may be 'OR'ed together in any way that makes sense. (Note that the significance of the numbers is as individual bits—in hex, for instance, a SnapBASIC file has its type byte set to 20H). Thus, to make an invisible Text file, you could do the following:

```
10 TEXT% = 8 : INVISIBLE% = 1
20 FOPEN "somefile"
30 POKE FADR(0)+2,
   BOR(TEXT%,INVISIBLE%)
```

and the next time you ran the File System, "somefile" would not appear on the menu (however, SnapBASIC could still access the file). This is a nice way to hide important information from the user. The Clock/Calender system, for example, creates an alarm file with the name CHR$(-1) (actually, hex 0FF) that is of types TEXT and INVISIBLE. With a little cleverness a SnapBA-SIC program could be generated that would create alarms and messages directly. Some judicious investigation of the format of the alarm file would be called for, however.

THE AUTO-OFF TIMER

The HHC maintains an Auto-Off timer that will automatically shut down if no input (in immediate mode or via the GET or INPUT statements) has been received. This can be disruptive if SnapBASIC has an intensely compute-bound program.

There are two ways to get around the Auto-Off timer;

1. Periodically "stoke" the timer so that it thinks a keystroke has occurred (even if there has been none). To do this, enter:

```
POKE 101, BOR(PEEK(101),64)
```

2. Disable the Auto-Off timer completely. To do this, enter;

```
POKE 101M BAND(PEEK(101),128)
```

Caution: do not POKE any other value into address 101.

CHAPTER 9: ASCII CHARACTERS

The character set used by the HHC is listed in order of the numbers that represent the characters in ASCII notation.

CONTROL CHARACTERS

The following table lists characters #0 through #31. These characters are control characters, rather than graphic characters; that is, their customary function is to perform a control function on an output device, rather than to display a symbol like 'A' or '?'.

The meanings of the columns in the table are:

- **Numeric value:** the number used to represent this character in the HHC's memory. If NV is the numeric value of the character, then CHR$I(NV) is the character.
- **HHC name:** the name, or description, used to identify this character in the HHC system.
- **ASCII name:** the name or description used to identify this character in "pure" ASCII.
- **HHC key:** key on the HHC keyboard that inputs this character.
- **HHC graphic:** the graphic symbol that represents this character when the character is displayed on the LCD. For characters that have a control function, you must use the "ESCDC" escape control sequence to display the character.
- **LCD action:** control function performed by this character when it is sent to the LCD. If this column is empty, the character has no control function; it displays the graphic symbol listed under "HHC graphic".

numeric value	HHC name	ASCII name
0		NUL, Null
1		SOH, Start of heading
2		STX, Start of text
3		ETX, End of text
4		EOT, End of transmission
5		ENQ, Enquiry
6		ACK, Acknowledge
7	Rotate; Bell	BEL, Bell
8	Backspace	BS, Backspace
9		HT, Horizontal tab
10	Line feed	LF, Line feed
11	I/O	VT, Vertical tab
12	Form feed	FF, Form feed
13	Enter	CR, Carriage return
14	Stop/Speed	SO, Shift out
15		SI, Shift in
16		DLE, Data link escape
17		DC1, Device control 1, XON
18		DC2, Device control 2
19		DC3, Device control 3, XOFF
20	Help	DC4, Device control 4
21	f1	NAK, Negative acknowledge
22	f2	SYN, Synchronous idle
23	f3	ETB, End of transmission block
24		CAN, Cancel
25		EM, End of medium
26		SUB, Substitute
27	Escape	ESC, Escape
28		FS, File separator
29		GS, Group separator
30		RS, Record separator
31		US, Unit separator

numeric value	HHC key	HHC graphic	LCD action
0		@	
1		A	
2		B	
3		C	
4		D	
5		E	
6		F	
7	ROTATE	G	"beep"
8		H	Backspace cursor; erase character under cursor after backspace.
9		I	
10		J	
11	I/O	K	
12		L	
13	ENTER	M	Erase LCD; move cursor to left edge.
14	STOP/ SPEED [1]	N	
15		O	
16		P	
17		Q	
18		R	
19		S	
20	HELP	T	
21	f1 [1]	U	
22	f2 [1]	V	
23	f3 [1]	W	
24		X	
25		Y	
26		Z	
27		[	Begins an escape control sequence.
28		\	
29		]	
30		^	
31		←	

[1] - This key has an immediate function in BASIC, and so cannot normally be read by GET.

DISPLAYABLE CHARACTERS

The HHC's use of characters #32 through #126 conforms exactly to the ASCII standard.

numeric value	HHC key	name
32	space	
33	!	exclamation mark
34	'	quotation mark
35	#	pound sign
36	$	dollar sign
37	%	percent sign
38	&	ampersand
39	'	apostrophe
40	(	left parenthesis
41	)	right parenthesis
42	*	asterisk, star, or 'times' sign
43	+	plus sign
44	,	comma
45	-	hyphen, dash, or minus sign
46	.	period
47	/	slash
48	0	
49	1	
50	2	
51	3	
52	4	
53	5	
54	6	
55	7	
56	8	
57	9	
58	:	colon
59	;	semicolon
60	<	left angle bracket or 'less than' sign
61	=	equal sign
62	>	right angle bracket or 'greater than' sign
63	?	question mark
64	@	
65	A	
66	B	
67	C	
68	D	
69	E	
70	F	
71	G	
72	H	
73	I	
74	J	
75	K	
76	L	
77	M	
78	N	
79	O	
80	P	
81	Q	
82	R	
83	S	
84	T	
85	U	
86	V	
87	W	
88	X	
89	Y	
90	Z	
91	[	left bracket
92	\	backslash
93	]	right bracket
94	^	caret
95	_	underscore
96	`	accent grave
97	a	
98	b	
99	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	

110	n	
111	o	
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	
123	{	left brace
124	|	vertical bar
125	}	right brace
126	˜	tilde

ADDITIONAL CHARACTERS

The HHC uses characters from #127 up as displayable characters and control characters. However, in SnapBASIC it is impossible to output characters above #127.

In "pure" ASCII, character #127 represents the control character "delete". Characters above #127 are undefined.

numeric value	HHC name	HHC key	HHC graphic	LCD action
127	"insert" cursor		▒	
128	up arrow	▲[2]	↑	
129	left arrow	◀[3]	←	Backspaces cursor; does not disturb character under cursor after move.
130	right arrow	▶[3]	→	Advances cursor; does not disturb character that was under cursor before move.
131	down arrow	▼[2]	↓	
132	"AM" symbol	INSERT[3]	A	
133	"PM" symbol	DELETE [3]	P	
134	superscript M	[4]	M	
135	division sign	[4]	÷	
136	"times" sign	[4]	×	
137	block cursor	SEARCH [2]	█	
138	"delete" cursor		▯	
139	"a" umlaut	C1[3]	ä	Except when read by GET, causes a "break" in execution of the current program.
140	"o" umlaut	C2 [5]	ö	
141	"u" umlaut	C3 [5]	ü	
142	"n" tilde	C4 [5]	ñ	

[2] - INPUT reads this character as ENTER.

[3] - INPUT cannot read this character; it performs its usual editing or control function.

[4] - INPUT skips this character.

[5] - INPUT is able to read these characters, but PRINT cannot print them.

CHAPTER 10: ERRORS

ERROR MESSAGE FORMAT

When BASIC detects a fatal error in immediate mode, it displays a message saying D 20

```
xx ERROR
```

where ***xx*** is one of the two-character codes described below.

When BASIC detects a fatal error in deferred mode, it displays a message saying D 20

```
xx ERROR IN nnnn
```

where ***xx*** is a two-character code and ***nnnn*** is the line number of the line that was being executed when the error occurred. After an error in a running program, all FOR . . . NEXT and GOSUB structures are exited.

FATAL ERROR CODES

AE -- Arithmetic Error

This can be due to : PB - a floating point number overflow PB - an illegal parameter to an arithmetic function (such as SQRT(-1) or LOG(-1))

AS -- Illegal Assignment

An assignment statement tried to assign a string value to a numeric variable, or vice versa; or an operator found a value of the wrong type; or a function found an argument of the wrong type.

AT -- Attach Unsuccessful

You attempted to attach a device, and could not for some reason (e.g., the device is not connected to the bus, there is not enough power to turn the device on, or there is not enough system memory to attach the device).

BU -- Burn error

You can not 'burn' the next statement. See BURN specifications. The program has not finished BURNing, but your source file is no longer useable. Delete whatever file remains, and make a new copy. Note that no line number is printed in this error message.

CC -- Can't Continue

You tried to CONTinue a program when you have not done a RUN, or after you edited the program.

CH -- Illegal Character

You have entered a character (such as a control character) that BASIC cannot interpret.

CO -- Command Error

You have attempted to execute a Deferred Mode Only command in immediate mode, or vice versa.

CX -- Too Complex Error

A line is too complex to compile; a line is too difficult to list; or you are using too many nested GOSUBs or FOR-NEXT loops, causing an overflow of the system stack.

DA -- Data Exhausted

You executed a READ statement, and no DATA items remained for it to read.

DF -- Undefined User Function (Run time error)

You tried to use a function that has not been defined.

DF -- Illegal User Function Name (Compile error)

User function name must start with FN.

FI -- File Error

A file error can occur in the following situations:

- file is not 'open'ed
- you tried to read from or write to a non existing record
- system memory is full; there is no room to expand or create the file

GO -- Undefined GOTO Statement

You tried to go to a non-existent line number with IF, GOTO, or GOSUB.

IQ -- Illegal Quantity

One of the following occurred:

- You tried to perform a calculation whose result was too large in magnitude to be represented in BASIC's numeric format. The largest number BASIC can represent is approximately 9.99999E1023. (Note that very small numbers, lesser in magnitude than 1.0E-1024, are reduced to zero.)
- You tried to divide a number by zero. Dividing zero by zero also produces this error.
- The quantity is too big to convert the real number to an integer.
- You tried to perform the ASC function on a null string.

NX -- NEXT without FOR

BASIC encountered a NEXT statement that did not correspond to a FOR/NEXT loop it was executing. This can be caused by a NEXT that does not match any FOR; a NEXT with the wrong variable name; or a GOTO that passes control to a line inside a FOR/NEXT loop without executing the FOR statement.

OD -- Outside Dimension

You tried to use an array element that is outside the dimensions of the array. This message can also occur if you use the wrong number of subscripts.

OM -- Out of Memory

There is not enough memory for your program to run. This can be caused by any combination of the following conditions: the program is too large; the program requires too much memory for variables, arrays, strings, and I/O conversions; or the memory specified by FREE is full.

OV -- Overflow

An arithmetic operation on integers has resulted in a value greater in magnitude than 32766.

RD -- Redimensioning Dimension

You tried to define an array with DIM after the array was already defined.

RT -- Return Without GOSUB

You tried to execute a RETURN without having executed a GOSUB. This is often caused by passing control to a subroutine with GOTO instead of GOSUB, or by letting control fall into a subroutine instead of passing it somewhere else.

SP -- Illegal Separator (Compile error)

BASIC expected a value (a variable, for example) but found a reserved word instead. For example; D 20

```
PRINT 12 + THEN
```

or D 20

```
A$ = GOTO$
```

SY -- System Error (Run time)

This error occurs if part of BASIC's memory has been overwritten by your program (possibly by POKEing) and LIST can no longer recognize it.

SY -- Syntax Error (Compile time)

You made an error in writing a statement, such as missing parentheses in an expression, use of a reserved word in a variable name, missing elements in a statement, a missing separator, etc.

TM -- Type Mismatch

You have attempted to perform an operation, but the operands are not of the correct type. Example: A$ * B$ (can't multiply strings).

UD -- Undimensioned Dimension

An array has to be dimensioned before you can access its elements. This error will also occur if you try to use a function that does not exist (since the syntax for a function reference and an array reference is identical.)

NON-FATAL ERRORS

****

When a number cannot be printed within the field specified by the parameters of the STRF$ function, a string of asterisks is returned, filling the entire field.

?

Returned by numeric to string conversion when a value that could not be represented by some number is encountered (possibly garbage was poked somewhere).

U

Returned in the integer portion of a number processed by STRF$ when an attempt is made to round more digits than the number has.

??

Displayed by the INPUT statement when the user entered fewer parameters than were requested, to request INPUT for the remaining variables only.

Rest Ignored

Displayed by INPUT when the user entered more variables than were requested, indicating that the extra values are lost.

Error, Retype Line?

Displayed by INPUT when an integer value has been requested and a non-number has been entered.

(Inverse video display)

When the LISTed line appears in inverse video, the expansion of the line by LIST has lost information (because it is longer than 80 characters).

CHAPTER 11: INTERNALS OF Snap-BASIC

This chapter provides information on the inner structure of SnapBASIC. The four SnapBASIC subsystems and the memory structure of SnapBASIC are described.

SnapBASIC SUBSYSTEMS

SnapBASIC can be thought of in terms of four seperate, somewhat independent operational modules: the Run Time Support Module (RTSM); the BASIC Compiler; the BASIC Decompiler and Lister; and the Eprom BURN module.

A) The Run Time Support Module has the responsibility for supporting the BASIC interpreter, and contains all of the run-time support code (organized as tags) needed for BASIC to execute that is not already part of the HHC support system.

B) The BASIC Compiler contains all of the code that supports the compiler actions. BASIC programs, consisting of numbered lines of BASIC source code, are compiled into a form that is supported by the RTSM. Every statement is compiled line-by-line into Snap tags following the format of the RTSM and the HHC support system, which are both based on the FORTH extension SnapFORTH. A line table (LT) is built, containing the information on where the compiled code is positioned in memory. At runtime, the compiled code is executed by Snap's inner Interpreter "NEXT" with no additional interpretive overhead.

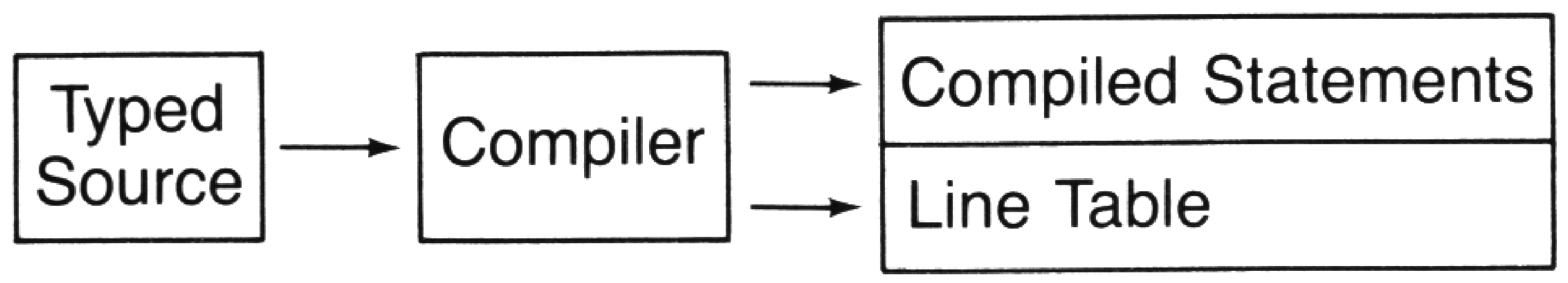

Figure 11-1

The actions of inserting, deleting, and changing lines are done through the Line Table, and change the pointers to the compiled code to reflect the new state of the program.

To RUN a program, the list of compiled statements are processed through the RTSM. Because of the structure of the Line Table, it is possible to start execution at any point in the program (by the RUN line-number command); it is also possible to STOP and CONTinue a program through the Line Table and the compiled statements.

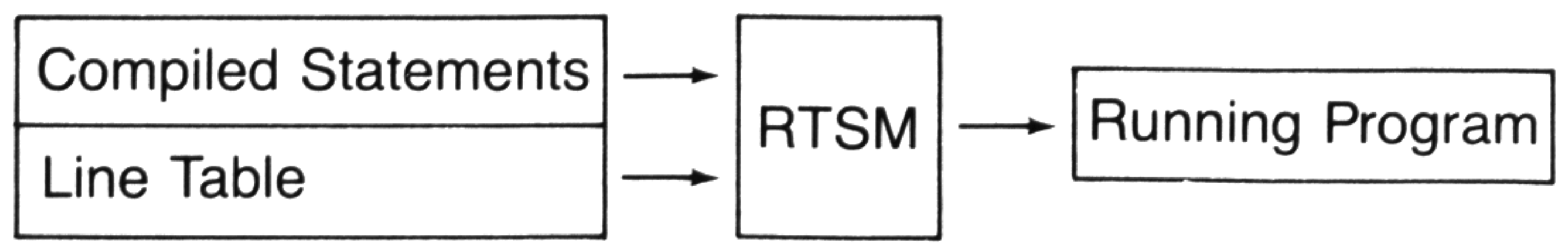

Figure 11-2

C) The BASIC Decompiler and Lister (BDL) contains all the code needed to decompile the compiled BASIC statements and to list the program in readable form. Compilation transforms BASIC's algebraic notation into Reverse Polish notation for execution by Snap; decompilation reconstructs the Reverse Polish to algebraic notation.

As the original source text is not saved by the system, it is not possible to LIST the orignal program text in the exact form it was entered. The BDL generates a recreation of the original text in ***intent***; i.e., in such a form that it reflects the original text in a more or less ***standard***, form.

The BDL is to be considered the inverse of the BASIC Compiler. It transforms the code segments in the compiled statements back to line numbers and BASIC words.

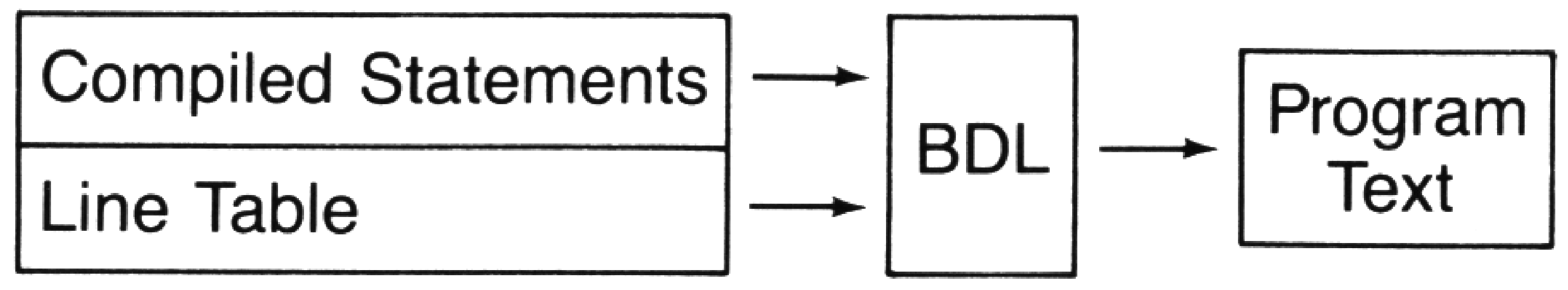

Figure 11-3

D) The BURN module contains all of the code needed to transform the compiled program to a form that can be executed out of a ROM capsule. To enable the program to execute without the availability of the RTSM, it is necessary to load and link any needed runtime support routines from the RTSM and to store them with the compiled program.

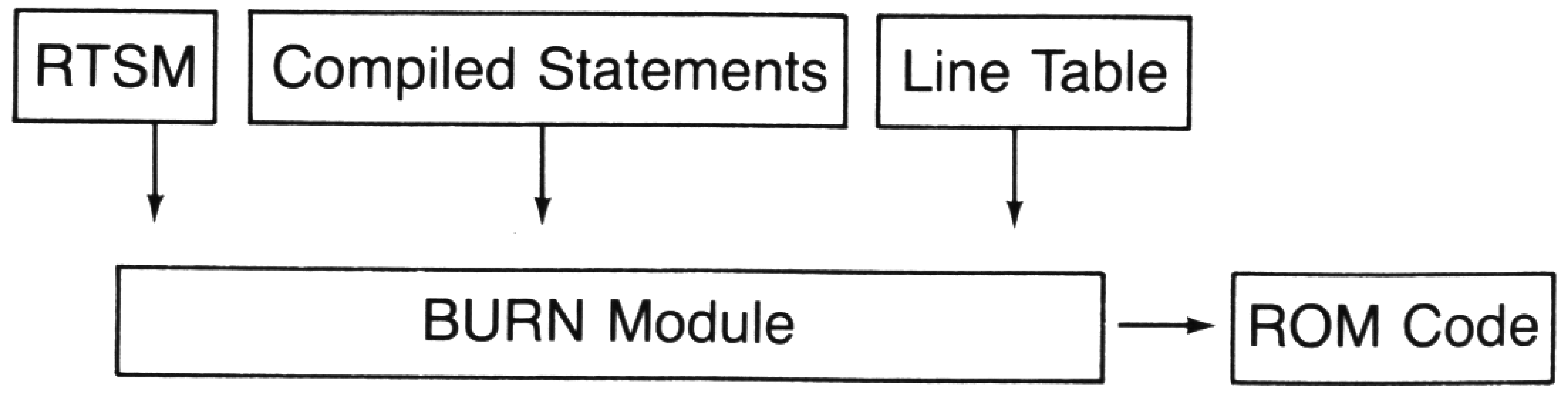

Figure 11-4

STRUCTURE OF SnapBASIC PROGRAMS BEFORE BURN

Figure 11-5 shows the structure of SnapBASIC programs before the BURN command is executed.

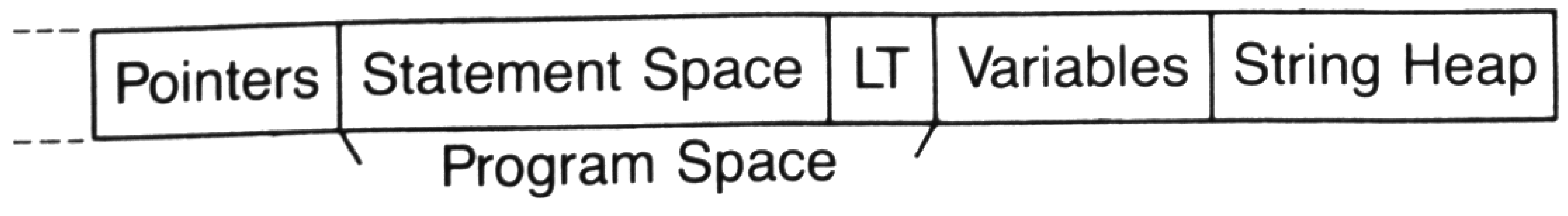

Figure 11-5

- The pointers are internal to the program space, and are relative to the file. They point to specific areas within BASIC. Examples of pointers are: pointers to a program, pointers to the line table, pointers to the variable names, pointers to user functions.
- The compiled statements are organized as one large continued "word" (from the Snap Interpreter's point of view); while the line table contains the address of each line within statement space. All statements are compiled in line number order. Jumps, such as GOTO, GOSUB, and so on, are compiled relative to the program space.
- The variables are discussed below.

The organization of the SnapBASIC program within the RAM file includes the following information:

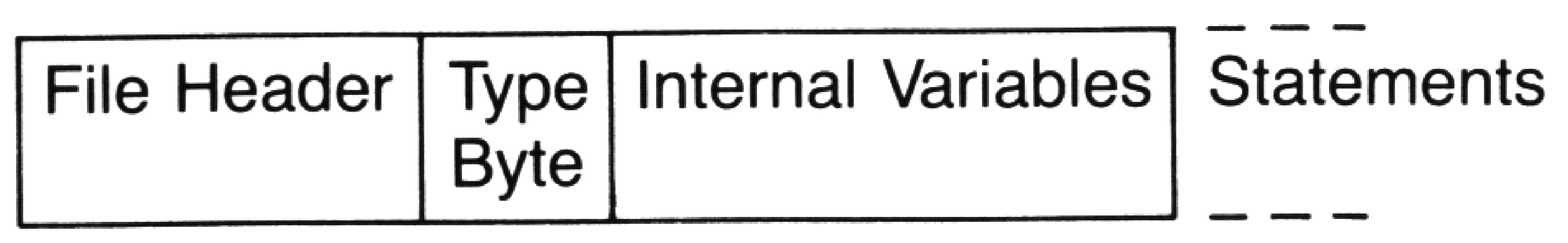

Figure 11-6

- The file header includes the name of the file
- The type designation is 20 Hex for a BASIC program file, and 02 Hex for a BASIC capsule image file.
- The internal variables are important variables in controlling the BASIC file. They are saved even when the CLEAR key is used, and are ***always*** updated on to the file. Examples of internal variables are: the number of program lines; go slow/go fast information {1}; long/short form program information {2}

{1} All jumps are compiled relative to the program, allowing changes in the program code. When the program is RUN, the program compiles absolute jumps for faster execution: all relative jumps are stored absolutely once computed and executed. Information is maintained to revert to slow mode for program changes, once BREAK or CLEAR is pressed.

{2} The CLEAR key keeps the long form of the program; BYE stores a short form produced by removing dimensional values; the String Heap and Variable values. When the program is selected from the BASIC menu, these areas of memory are allocated again.

STRUCTURE OF VARIABLES

Variables are named, and are of several different types: Integer, Real, String, and Boolean. Arrays of variables are also named, and consist of the same types as the simple variables. Array dimensions are put with the names.

Var Names	Var Values	String Heap→	←Dimensional Values

Figure 11-7

- The variable values are associated with the names, and include the values of simple variables.
- The string heap contains all strings, and is allocated dynamically. A garbage collector runs automatically to restore free space when necessary.
- The dimensional values include the (dynamic) arrays.
- Together, the string heap and the dimensional values fill up the memory that is available for BASIC within the current RAM area.

STRUCTURE OF THE LINE TABLE

The line table keeps track of the position of all the BASIC lines in memory. The line table is deleted from the BURNed program. The format of the line table is:

1) Relative pointer to the beginning of the line (2 bytes)
2) Line number (2 bytes)

STRUCTURE OF THE VARIABLE NAMES

The variable names are constructed as sequences of strings of characters for names of the four simple types and four array types. The names of simple variables are represented as follows:

xcccc...ccc***x***cc...ccc***x***ccc... etc.

The names of dimensioned arrays are represented as:

xccc...ccc***x***#parms,1-dim,2-dim,...,n-dim***x***ccc... etc.

where:

- ***x*** represents a distinguishable start byte, indicating the beginning of a new item
- c represents the characters of the name
- #parms is the number of dimensions
- i-dim represents the size of the i-th dimension

STRUCTURE OF THE DIMENSION STACK

The string heap consists of a string of characters. The dimension stack consists of a series of arrays, each of which has the following format (note that this grows downward in memory, toward the top of the string heap):

# dimensions	2 bytes
i-dim	2 bytes/dimension
bytes/cell	2 bytes (as in the table just above)
values	bytes/cell bytes each

MEMORY USED TO STORE VARIABLES

Memory used for variable storage is as follows:

integers	2 bytes
reals	8 bytes
booleans	1 byte
strings	2 byte pointer to string heap
arrays	2 byte relative pointer to dimension stack

INDEX

FRIENDS AMIS, INC.

An important note: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or programs are satisfactory.