

Panasonic



MICROSOFT BASIC
VOLUME II: REFERENCE GUIDE

A PROGRAMMING LANGUAGE
FOR THE HHC^{T.M.}

RL-S6001S7

A vertical spiral binding runs down the center of the page, consisting of a series of dark, circular loops.

MICROSOFT BASIC

a programming language
for the HHC^{T.M.}

VOLUME II: REFERENCE GUIDE

by
FRIENDS AMIS, INC.

TABLE OF CONTENTS

CHAPTER 1: STATEMENTS AND COMMANDS

?	1-1
ASSIGNMENT	1-2
ATTACH	1-3
BYE	1-3
CALL	1-4
CLEAR	1-4
CONT	1-4
DATA	1-5
About String Values	1-5
DEF	1-6
DIM	1-7
END	1-7
FOR	1-8
Terminology	1-8
Note On Ending a Loop	1-8
Notes On NEXT	1-8
No-Nos	1-9
GET	1-10
GOSUB	1-11
GOTO	1-11
IF	1-12
INPUT	1-12
LET	1-13
LIST	1-14
Interacting With LIST	1-14
Note On Deferred Execution	1-14
NEW	1-15
NEXT	1-15
ON	1-16
POKE	1-16
PRINT	1-17
How Values Are Positioned	1-17
PRINT With No Values	1-17
Miscellaneous Notes	1-18
READ	1-18
REM	1-19
RESTORE	1-20
RETURN	1-20
STOP	1-20
TROFF	1-21
TRON	1-21

CHAPTER 2: INTRINSIC FUNCTIONS

ARITHMETIC FUNCTIONS	2-1
STRING FUNCTIONS	2-2
FUNCTIONS THAT MAY BE USED ONLY AS PRINT VALUES	2-4
TRIGONOMETRIC FUNCTIONS	2-4
The Functions	2-4
Constants and Variables	2-5
Separability	2-6

CHAPTER 3: OPERATORS

NUMERIC OPERATORS	3-1
STRING OPERATORS	3-2

CHAPTER 4: RESERVED WORDS

CHAPTER 5: LINE NUMBERS, NUMBERS, AND STRINGS

LINES	5-1
NUMERIC VALUES	5-1
INTEGER VALUES	5-1
RULES FOR DISPLAYING NUMERIC VALUES (PRINT AND STR\$)	5-1
RULES FOR READING NUMERIC VALUES (INPUT AND VAL)	5-2

CHAPTER 6: BASIC PROGRAM EDITOR

QUICK REFERENCE

ADDING, DELETING, REPLACING, OR COPYING A LINE	6-1
CURSOR MOTION	6-1
INSERT CHARACTERS	6-2
DELETE CHARACTERS	6-2
SHORTCUTS	6-2

CHAPTER 7: PERIPHERAL DEVICES

CONTROL CHARACTERS	7-1
ASCII Control Characters	7-1
ESCAPE CONTROL SEQUENCES	7-2
Opcodes	7-2
THE KEYBOARD	7-5
THE LCD	7-5
THE TV ADAPTOR	7-6
THE MICRO PRINTER	7-6

SERIAL INTERFACE ADAPTOR	7-7
Initializing the Serial Interface	7-7
Note About Protocols	7-8
THE MODEM	7-10
Initializing the Modem	7-11

CHAPTER 8: PEEKS AND POKES

INPUT AND OUTPUT	8-1
The System Device Table (SDT)	8-1
The ATTACH Statement	8-2
THE HELP AND I/O KEYS	8-2
THE KEYBOARD BUFFER	8-3
Structure Of the Keyboard Buffer	8-3
The Pushkey Buffer	8-4
Buffer Locations	8-5
PEEKs and POKEs	8-5
FUNCTION KEYS	8-6
"Typing" a Function Key	8-6
THE STOP/SPEED KEY	8-7
DATE AND TIME	8-7
Format Of the Date and Time	8-8
ROTATION MODE	8-9
VARIABLES AND ARRAYS	8-10
Locations of Variables and Arrays	8-10
Formats of Variables	8-11
Formats of Arrays	8-12
Format Of a Numeric Value	8-13
Other Useful Addresses	8-13
Finding a Variable's Value	8-14

CHAPTER 9: ASCII CHARACTERS

CONTROL CHARACTERS	9-1
DISPLAYABLE CHARACTERS	9-3
ADDITIONAL CHARACTERS	9-5

CHAPTER 10: ERRORS

ERROR MESSAGE FORMAT	10-1
ERROR CODES	10-1

PREFACE

This book is a reference guide to use when programming with Microsoft Basic on your HHC™. It provides detailed information that you can refer to as you write programs.

The companion volume to this manual is the *Microsoft Basic Tutorial*, which teaches you how to program in Microsoft Basic.

If you are familiar with the HHC, but not with Basic, you can skim over the sections of the book that are marked with {H}.

If you are familiar with both Basic and the HHC, and only need to know how Microsoft Basic on the HHC differs from other Basics, you can skim over most of the book; pay attention to the sections that are marked with {B}.

This book presents examples of the characters exactly as they appear on the HHC display. We have used the ► symbol to indicate when a one-line display is continued on the following line in the example.

CHAPTER 1: STATEMENTS AND COMMANDS

Below is a complete description of every statement and command that Microsoft Basic 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.”

n	represents a numeric value (constant, variable, expression, etc). ⁽¹⁾
nv	represents a numeric variable or array element.
nc	represents a numeric constant.
s	represents a string value.
sv	indicates a string variable.
sc	represents a string constant.
x	represents a value of either numeric or string type.
v	represents a variable of either numeric or string type.
ln	represents a line number.

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

?

{B}

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

Examples:

```
? MI
```

```
? "Fuel efficiency=";MI;"mpg."
```

⁽¹⁾ - Integer and numeric values are interchangeable, except where noted otherwise.

```
? "Order=;LEN (OS#)-3
```

```
? I;TAB (5);
```

```
? #2,I;TAB (5);
```

ASSIGNMENT

Format: [LET] *nv* = *n*

or [LET] *sv* = *s*

The value of the expression *n* or *s* is assigned to the variable *nv* or *sv*.

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:

```
X=5
```

```
LET X=5
```

```
X=X+1
```

```
XS=5*(X*(N)^2+ABS (Y))
```

```
ST#='All systems go'
```

```
ST#=ST#+', '
```

```
TR#=STR$(LEN (ST#))+', '+ST#
```

ATTACH

{B}

Format: ATTACH *n1* TO #*n2*

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

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.

Examples:

```
ATTACH 68 TO #2
```

```
ATTACH UN TO #2+BA
```

```
ATTACH IT(N) TO #3
```

BYE

{B}

Format: BYE

BYE returns you from the Basic interpreter, which allows you to edit and run a program, to the Basic menu, which allows you to choose a program to edit and run, and to use the I/O and HELP keys.

Always use BYE to return from the Basic interpreter to the Basic menu. If you use the CLEAR key, the HHC's memory area will appear to be full, and you will be unable to run or edit any Basic program.

If you accidentally use CLEAR to leave the Basic interpreter, use the following procedure to restore the file system:

1. Return to the primary menu.
2. Select the file system.
3. Delete the file named B from intrinsic RAM (select B from the file system's menu and delete its name).
4. Leave the file system and select Basic from the primary menu.
5. Select the file you were editing when you pressed CLEAR.
6. Save the file with BYE.

{B} CALL

Format: CALL *n*

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

n is the address of the subroutine's entry point in the HHC's memory. *n* must be in the range 0 to 65535.

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

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

{B} CLEAR

Format: CLEAR

The CLEAR statement does the following things:

- Sets all numeric variables to zero and all string variables to null.
- Erases the index, step and limit of any active FOR/NEXT loop, and the return pointer of any active GOSUB.
- 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

{B} Use CONT to restart a program after you have interrupted it by pressing the C1 key, or after it has executed a STOP or END.

You may display and change the values of variables while the program is interrupted. Except for the changes you make, 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 an error has occurred; after you have edited the program; or before you have begun RUNing the program.

DATA

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.

DATA is legal in immediate mode, but it has no purpose there.

About String Values

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

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

A string value containing lower case letters must also be in {B} quotes, or Basic will shift the letters to upper case when it stores the DATA statement.

A string value with leading or trailing blanks need *not* be in quotes. READ does not trim the blanks when it reads the value.

A quotation mark that appears after the first character in a string value is treated as a character in the value like any other.

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 FN *mm*(*pr*)=*ex*

DEF defines a function named *mm*. *mm* may be any name that would be valid as the name of a numeric variable. (But the name of a function is distinct from the name of a numeric variable or array; you can use the same name for both, and Basic will always know the difference by context.)

pr is the function's formal parameter. It may be any name that would be valid as the name of a numeric variable. (But the names of a numeric variable and a formal parameter are distinct.)

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, Basic does the following:

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

Examples of definitions:

```
DEF FN A(X)=X^2
```

```
DEF FN AR(Q)=Q*(Q+1)
```

```
DEF FN YY(Z)=4+LEN (ST$)
```

Examples of use:

```
PRINT FN A(X)
```

```
PRINT X:FN A(X-2)
```

```
X=FN YY(0)
```

DIM

Format: DIM xa(*n1* [, *n2*, . . . *nx*])
or DIM xa\$(*n1*[, *n2*, . . . *nx*])

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

Array elements are numbered from 0; thus, the array's elements are numbered 0 to *n1*-1 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.

If you refer to an array without first defining it with DIM, Basic automatically defines it as an array with one dimension and 10 elements.

Examples:

```
DIM X(10)
```

```
DIM XN$(8,2)
```

```
DIM Y$(N),YN$(N)
```

END

Format: END

END terminates execution of your program. Unlike STOP, it does not display a "Break in *nnnn*" message.

Examples:

```
END
```

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

FOR

Format: FOR $nv = n1$ TO $n2$ [STEP $n3$]

FOR begins a FOR/NEXT loop.

Basic executes the statements between 'FOR $nv = \dots$ ' and 'NEXT x ' with $nv = n1$, then with $nv = n1 + n3$, then with $nv = n1 + 2 \cdot n3$ etc. Basic stops looping after the last pass for which $nv \leq n2$.

If $n3$ is absent, Basic assumes $n3 = 1$.

If $n3 < 0$, Basic stops the loop after the last pass for which $nv \geq n2$.

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 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 be the value it had the last time through the loop, *plus* the step.

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

Notes On NEXT

You may end two or more loops at the same point by putting both of their subscripts in the same NEXT, *innermost subscript first*, like this:

```
450 FOR I=1 TO 10      begins outer loop
460 FOR J=3 TO 8      begins inner loop
.
.
530 NEXT J,I          ends both loops
```

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
```

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. (It works in this Basic, but may not work in others, and it is generally a bad practice.)
- Using an integer variable for the index (you'll get an SN error).

```
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)
```

{B} GET

Format: GET [#n,]sv
or GET [#n:]sv

The two forms of GET shown above are completely equivalent.

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

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

If the LUN is #0, Basic does not echo the character on the LCD, as it would for INPUT.

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

Note: there are two restrictions on your use of the keyboard when you are using GET. First, GET does not "read" function key definitions correctly. It misses a keystroke for every program line that is executed between one GET and the next. Second, GET does not read keyboard characters correctly if you "type ahead" of the program; again, it misses a keystroke for every program line that is executed between one GET and the next while keystrokes are waiting to be read.

Examples:

```
GET CH#
```

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

```
GET #CH:CH#
```

GOSUB

Format: GOSUB In

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

Example:

```
GOSUB 1010
```

GOTO

Format: GOTO In

Transfers control to the first statement on line number *In*.

Example:

```
GOTO 750
```

IF

Format: IF logical-expression GOTO *ln*
or IF logical-expression THEN *stmt*[:*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.

If **logical-expression** has a non-zero value, IF does a GOTO to *ln*, or executes each **stmt** from THEN to the end of this line.

If **logical-expression** has a zero value, IF does nothing.

Examples:

```
IF I<10 GOTO 950
```

```
IF I>INT (I)THEN I=INT (I)+1
```

```
IF ER THEN GOSUB 2010
```

```
IF I<>INT (I)THEN PRINT "Must be an integer." : GOTO 950
```

INPUT (deferred mode only)

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, Basic assumes LUN #0 (the keyboard).

"prompt" is a prompting message. If it is present, INPUT prompts the user with this message, suffixed with a '?'. If **"prompt"** is absent, INPUT prompts the user '?'.
"prompt" must be a string **constant**. If you use a string variable, INPUT will try to read into the variable. If you use an

expression, you will get an SN error.

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

INPUT reads values into the variables in the order that the variables appear in the statement. A value may be terminated by a comma 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. {B}

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. Note that this is different from the behavior of READ, which saves the extras for the next READ.

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.

{B} LIST

Format: LIST

or LIST #n

or LIST ln

or LIST #n,ln

LIST begins listing the program you are editing at line *ln*. If *ln* is not given, LIST begins listing the program at its first line.

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

Interacting With LIST

When you enter LIST and press RETURN, LIST displays the first line you have requested. (If you are listing on a device other than the LCD, you may have to press \blacktriangleright to see the first line.)

If the 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.

To see the second line, press \blacktriangleright . To see the third line, press \blacktriangleright again, and so on. When you press \blacktriangleright after seeing the last line, LIST ends.⁽²⁾

To end LIST at any point, press ENTER instead of \blacktriangleright .

To edit a line, LIST the line and use the editing keys (see Chapter 6) as necessary. When you are done editing the line, press ENTER. LIST stores the changed line in your program and ends. You must use LIST again to see and edit more of your program.

If you begin editing a line and then decide not to change it after all, press the \blacktriangleright key. Basic will list the next line *without* storing the changed line in your program.

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.

⁽²⁾ - Unlike the HHC'S file system editor, the Basic editor only lets you move down in a file. You cannot use the \blacktriangleleft key to move up.

NEW

Format: NEW

NEW does the following things:

- Sets all numeric variables to zero and all string variables to null.
- Erases the index, step and limit of any active FOR/NEXT loop, and the return pointer of any active GOSUB.
- Deletes all the lines from the program.

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

NEXT (deferred mode only)

Format: NEXT

or NEXT nv

or NEXT nvx, . . . ,nv2,nv1

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. If several *nv* 's are used on one NEXT, the one closing the innermost loop must be first.

For more information, see the description of FOR.

```
NEXT
```

```
NEXT I
```

```
NEXT K,J,I
```

ON

Format: ON n GOTO ln1,ln2,...,lnx
or ON n GOSUB ln1,ln2,...,lnx

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 *1n1*. If *n*=2, ON does a GOTO or GOSUB to *1n2*, and so forth.

Examples:

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

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

POKE

Format: POKE n1,n2

Stores *n2* in the HHC's memory at the memory address *n1*. *n1* must be in the range 0 to 65535. *n2* must be in the range 0 to 255.

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 *Microsoft Basic Tutorial Guide* for more information.

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

PRINT

Format: $\left. \begin{array}{l} \text{PRINT} \\ ? \end{array} \right\} \left\{ \begin{array}{l} [\#n,] x1 \{,l\} x2 \{,l\} \dots xy \{,l\} \\ [\#n] \end{array} \right\} \{B\}$

'?' is an abbreviation for PRINT. If you enter '?' in a statement, Basic will display 'PRINT' when you LIST the statement.

PRINT writes information to the LUN (logical unit number) specified by *n*. If *n* is absent, Basic 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 16 characters long. The first *x* is displayed at the beginning of the first zone. {B}

If the first and second *x*'s are separated by a semicolon, Basic puts no spaces between them. (But recall that a space is displayed at the end of a numeric value, as part of the value.) If the first and second *x*'s are separated by a comma, Basic skips to the beginning of the next zone before displaying the second value. {B}

If the last *x* is followed by a semicolon, Basic 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, Basic 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, Basic ends the line, so that the next PRINT statement will begin writing data at the start of the next line.

Basic has two special functions, SPC and TAB, which may be used only as PRINT statement expressions. SPC (N) inserts N spaces in the output line. TAB (N) advances the cursor to column N; if the cursor is already at or past column N, TAB does nothing.

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.

Miscellaneous Notes

The character that separates the LUN from the first *x* may be either a comma or a semicolon. The two are equivalent in this context. Since they come before the first *x*, they have no effect on the output.

Examples:

```
PRINT MI
```

```
PRINT "Fuel efficiency=";MI;"MPG."
```

```
PRINT "Order=";LEN (OS$)-3
```

```
PRINT I;TAB (5);
```

```
PRINT #2,I;TAB (5);
```

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, . . .

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.

Examples:

```
READ N
```

```
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.

Basic inserts a space between REM and the beginning of the remark. Thus, if you enter this,

```
2020 remvalues within limits?
```

Basic will store this:

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

and if you enter this,

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

Basic will store this:

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

(with two blanks after the REM).

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

Examples:

```
REM If array still empty, give error.
```

```
REM Input: user is PromPted for data.▶  
from coding sheet.
```

```
REM Copyright 1981 by Consolidated▶  
EnterPrise Corp.
```

```
REM Verify that ABS (SOR (A^2+B^2))▶  
-1<.5*10^-3.
```

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.

STOP

Format: STOP

Interrupts execution of the program and displays the message

Break in nnnn

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

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

TROFF

{B}

Format: TROFF

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

TRON

{B}

Format: TRON

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

```
Ln1Ln2Ln3 . . .
```

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 by TROFF, or until you return to the Basic menu with BYE.

CHAPTER 2: INTRINSIC FUNCTIONS

ARITHMETIC FUNCTIONS

$nv = \text{ABS}(n)$

Returns the absolute value of n .

$nv = \text{EXP}(n)$

Returns the constant E (2.71828183) raised to the power n . The maximum value of n that will not produce an overflow error is 88.02969.

$nv = \text{FRE}(n)$

{B}

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. n is ignored.

$nv = \text{INT}(n)$

Returns the largest integer nv such that $n \leq nv$. $\text{INT}(1.1)$ is 1; $\text{INT}(1)$ is 1; $\text{INT}(.9)$ is 0; $\text{INT}(-.1)$ is -1.

$nv = \text{LOG}(n)$

Returns the the natural (base E) log of n . To obtain the base Y log of X, use the formula $\text{LOG}(X)/\text{LOG}(Y)$.

Example: the base 10 (common) log of 7 is $\text{LOG}(7)/\text{LOG}(10)$.

$nv = \text{PEEK}(n)$

Returns the value stored at memory address n . n must be in the range 0 to 65535. nv is in the range 0 to 255.

$nv = \text{POS}(n)$

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 printable character (code #32 or greater) adds 1 to the value POS will return. PRINTing a control character has no effect on the value.

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.

nv = RND (n)

Returns a "random" number in the range $0 \leq \text{RND}(n) < 1$.

The first time you call RND, use a negative *n*. 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 a positive *n*. This makes RND return the next random number in the sequence.

Using a zero value for *n* makes RND return the same random number it returned the preceding time it was called. This is sometimes more convenient than saving the last value in a variable.

nv = SGN (n)

Returns the sign of *n*: 1 if $n > 0$, 0 if $n = 0$, -1 if $n < 0$.

nv = SQR (n)

Returns the square root of *n*. Equivalent to $n^{.5}$. Causes an "FC error" ("illegal quantity error") if $n < 0$.

SQR(N) produces the same result as $N^{.5}$, but executes more quickly.

STRING FUNCTIONS

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 gives an FC error.

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 '!'.
If $n < 0$ or $n \geq 256$, CHR\$ gives an 'FC error'.

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 $n > \text{LEN}(s)$, LEFT\$ returns *s*.

If $n < 0$ or $n \geq 256$, LEFT\$ gives an 'FC error'.

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 in *s*; and so forth.

If *n1* is not an integer value, MID\$ truncates it. If $n1 \geq \text{LEN}(s)$, MID\$ returns the null string. If $n1 < 0$ or $n1 \geq 256$, MID\$ gives an 'FC error'.

If *n2* = 0, MID\$ returns the null string. If *n2* is greater than number of characters remaining in the string from character *n1* to the end, MID\$ returns the entire part of the string from character *n1* to the end. If $n2 < 0$ or $n2 \geq 256$, MID\$ gives an 'FC error'.

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.

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.

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 an invalid character in a number expressed in scientific notation. For example, '2.5E3' is valid; '2.5e3' is invalid, and will return the value 2.5.

FUNCTIONS THAT MAY BE USED ONLY AS PRINT VALUES

SPC (n)

Prints *n* spaces.

If $n < 0$ or $n \geq 256$, SPC gives an 'FC error'.

TAB (n)

Inserts enough spaces before the following value to make the value begin in column *n*. If the displayed line is already filled up to or beyond column *n*, TAB does nothing.

If $n < 0$ or $n \geq 256$, TAB gives an 'FC error'.

TRIGONOMETRIC FUNCTIONS

Microsoft Basic on the HHC does not have built-in trigonometric functions. You can use the following subroutines to calculate trigonometric functions if you need them.

These subroutines are based on procedures described in *Software Manual for the Elementary Functions*, by Cody and Waite.⁽¹⁾

The Functions

The trigonometric functions implemented by these subroutines are: SIN, COS, TAN, COTAN, ATAN (arctangent) and ATAN2 (another version of arctangent).

⁽¹⁾ - Software Manual for the Elementary Functions, Cody, William J., Jr., and Waite, William. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980.

SIN, COS, TAN and COTAN expect an angle, in radians, in the variable TX. They return the result in TR.

The functions reduce large angles to the smallest possible range before evaluating a function. Very large angles cause the functions to lose precision, since such large numbers cannot be represented as precisely as small ones. See Cody and Waite for details.

ATAN expects a scalar input value in TX. It returns its result, in radians, in TR.

ATAN2 expects two inputs in TU and TV. It returns its result, in radians, in TR. ATAN2 calculates the angle between the positive *u*-axis and the directed line segment to the point (TU,TV).

If TU is zero and TV is non-zero, ATAN2 is defined to be $\pm\pi/2$.

Constants and Variables

The subprograms use variables beginning with the letter K to refer to constant values. This speeds up execution significantly, since interpreting numeric constants is Microsoft Basic's slowest operation by a large margin.

All "non-constant variables" in the subprograms begin with the letter T.

Most of the "constant variables" are preceded by remarks that list the corresponding values in the Cody and Waite algorithms. Note that Cody and Waite sometimes use the same names, such as C1, C2, C3, ..., for different constants in different pairs of functions. The subprograms store these constants in distinct variables.

The table below shows how the names of "non-constant variables" correspond to the names used in Cody and Waite's flowcharts.

SIN and COS

<u>variable</u>	<u>use</u>
TF	f
TN	XN
TR	g, R(g) & result
TS	SGN
TT	--
TX	X
TY	Y

TAN and COTAN

<u>variable</u>	<u>use</u>
TF	f
TN	XN
TR	result
TS	XNUM
TT	g & XDEN
TX	X

ATAN and ATAN2

<u>variable</u>	<u>use</u>
TF	f; exponent of TV
TN	N; exponent of TU; internal-format exponent of most numeric variable most recently referred to
TR	result
TU	U
TV	V
TX	X
TY	g & R(g)

Separability

The subprograms are divided into three groups: SIN/COS, TAN/COTAN, and ATAN/ATAN2. Each group is independent of the others, and may be used alone.

The initialization section is sub-divided by the subprograms that the constants serve. If you include only one or two of the subprograms in a given program, you may include only the constants used by those subprograms. **But note** that some constants serve more than one pair of functions; these constants are listed separately from the others.

```

1100 FCS = " FC ERROR"
1200 REM ***FOR ALL FUNCTIONS***
1210 REM EPS
1220 KF = 2E - 16
1230 REM ***SIN/COS & ATAN/ATAN2***
1240 REM PI/2
1310 KA = 1.57679633
1320 REM ***SIN/COS***
1330 KB = INT (3.14159265 * 2 ^ 16)
1340 REM 1/PI
1350 KC = .318309886
1360 REM C1

```

```

1370 KD = 201 / 64
1380 REM C2
1390 KE = .967653590E - 3
1400 REM R1,R2,R3,R4,R5.
1410 DIM KK(5)
1420 KK(0) = - .166666666:KK(1) = .833333072E - 2
1430 KK(2) = - .198408328E - 3:KK(3) = .275239711E - 5
1440 KK(4) = - .238683464E - 7
1500 REM ***TAN/COTAN***
1510 REM YMAX
1520 KG = INT (2 ^ 16 * 3.14159265 / 2)
1530 REM 2/PI
1540 KH = .636619772
1550 REM C1
1560 KI = 201 / 128
1570 REM C2
1580 KJ = .000483826795
1590 REM EPSI
1600 KL = .85E38
1610 REM ***ATAN/ATAN2***
1620 REM 2-SQR(3)
1630 KM = .267949192
1640 REM SQR(3)
1650 KN = 1.7320508
1660 KP = 3.14159265
1670 REM A0,A1,A2,A3
1680 DIM KQ(4)
1690 KQ(0) = 0:KQ(1) = .523598776
1700 KQ(2) = 1.57079632:KQ(3) = 1.04719755
1710 REM P1
1720 KR = - .720026849
1730 REM P0
1740 KS = - 1.4400834
1750 REM Q1
1760 KT = 4.75222585
1770 REM Q0
1780 KU = 4.3202504
1790 REM ***TAN***
1800 REM P1
1810 KV = - .111361440
1820 REM P2
1830 KW = .00107515474
1840 REM Q0
1850 KX = 1.
1860 REM Q1
1870 KY = - .444694772
1880 REM Q2
1890 KZ = .0159733921
1900 STOP
10000 REM ***TR = SIN(TX)***

```

```

10010 TY = ABS (TX):TS = SGN (TX): GOTO 10050
10020   REM ***TR=cos(TX)***
10030 TY = ABS (TX) + KA:TS = 1: GOTO 10050
10040   REM COMMON TO SIN & COS
10050   IF TY > = KB THEN PRINT "SIN/COS";FC$: STOP
10060 TN = INT (TY * KC + .5)
10070 TT = .5 * TN: IF TT > INT (TT) THEN TS = - TS
10080   IF ABS (TX) < > TY THEN TN = TN - .5
10090 TF = ( ABS (TX) - TN * KD) - TN * KE
10100   IF ABS (TF) < KF THEN TR = TF: GOTO 10130
10110 TR = TF * TF
10120 TR = (((KK(4) * TR + KK(3)) * TR + KK(2)) * ►
TR + KK(1)) * TR + KK(0)) * TR
10130 TR = TF + TF * TR
10140   IF TS < 0 THEN TR = - TR
10150   RETURN
11000   REM ***TR=tan(TX)***
11010   GOSUB 11100
11020   IF TN = INT (TN) THEN TR = TS / TT: RETURN
11030 TR = - TT / TS: RETURN
11040   REM ***TR=cotan(TX)***
11050   IF ABS (TX) > KL THEN PRINT "COTAN";FC$: STOP
11060   GOSUB 11100
11070   IF TN = INT (TN) THEN TR = TT / TS: RETURN
11080 TR = - TS / TT: RETURN
11100   REM COMMON TO TAN & COTAN.
11110   IF ABS (TX) > KG THEN PRINT "TAN/COTAN";►
FC$: STOP
11120 TN = INT (TX * KH + .5)
11130 TF = (TX - TN * KI) - TN * KJ
11140   IF ABS (TF) < KF THEN TS = TF:TT = 1: GOTO 11180
11150 TT = TF * TF
11160 TS = (KW * TT + KV) * TT * TF + TF
11170 TT = (KZ * TT + KY) * TT + KX
11180 TN = .5 * TN: RETURN
12000   REM ***TR=atan(TX)***
12010 TF = ABS (TX): GOSUB 12200
12020   IF TX < 0 THEN TR = - TR
12030   RETURN
12040   REM ***TR=atan2(TU,TV)***
12050   IF TU < > 0 GOTO 12080
12060   IF TV = 0 THEN PRINT "ATAN2";FC$: STOP
12070 TR = KA: GOTO 12130
12080 TN = TV: GOSUB 12310:TF = TN
12090 TN = TU: GOSUB 12310
12100   IF TF - TN > 126 THEN TR = KA: GOTO 12120
12110 TF = ABS (TV / TU)
12120   IF TF = 0 THEN TR = 0: GOTO 12140
12130   GOSUB 12200
12140   IF TU < 0 THEN TR = KP - TR

```

```

12150   IF TV < 0 THEN TR = - TR
12160   RETURN
12200   REM COMMON TO ATAN & ATAN2. IN=TF, OUT=TR.
12210 TN = 0
12220   IF TF > 1 THEN TF = 1 / TF:TN = 2
12230   IF TF > KM THEN TF ((TF * KN) - 1) / (KN + TF):TN = ►
TN + 1
12240   IF ABS (TF) < KF THEN TR = TF: GOTO 12280
12250 TY = TF * TF
12260 TY = ((TY * KR + KS) * TY) / ((TY + KT) * TY + KU)
12270 TR = TF + TF * TY
12280   IF TN > 1 THEN TR = - TR
12290 TR = KQ(TN) + TR: RETURN
12300   REM TN=EXP. OF MOST RECENTLY REFERENCED ►
VARIABLE
12310 TN = PEEK ( PEEK (133) + 256 * PEEK (134))
12320   RETURN

```

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

operator	priority	function	example of use	result
()	0	overrides other priority rules	$A = 5 \cdot (4 + 3)$	$A = 35$
-	1	negation	$A = -8 \wedge 2$	$A = 64$
\wedge	2	exponentiation	$A = 2 \wedge 3$	$A = 8$
*	3	multiplication	$A = 2 \cdot 6$	$A = 12$
/	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		
NOT	6	logical "not" ^{1}	IF NOT 6=5 GOTO 90 A=NOT-1	go to 90 A=0
AND	7	logical "and" ^{1}	IF 1 AND 0 GOTO 90 A=1 AND 1	no action A=1
OR	8	logical "or" ^{1}	IF 1 OR 0 GOTO 90 A=0 OR 0	goes to 90 A=0

^{1}- Logical operators act on integer values rather than numeric values. Integer values must be in the range -32768 to +32767. A value out of this range causes an FC error.

STRING OPERATORS

String Operators				
operator	priority ⁽²⁾	Function	example of use	result
()	0	overrides other priority rules ⁽²⁾		
+	4	concatenation	A\$ = '8' + '1'	A\$ = '81'
=	5	is equal to		
<>	5	is not equal to	IF 'XX' = 'X' GOTO 90	no action
<	5	is less than	FL = 'XX' <> 'X'	FL = -1
<=	5	is less than or equal to		
>	5	is greater than		
>=	5	is greater than or equal to		

Where Basic 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 Basic evaluates 'X\$ = Y\$' first, yielding a 0 or -1, which may be added to A.

⁽²⁾- There are no situations in Microsoft Basic where parentheses may be used to alter the order of precedence in a string expression; however, parentheses are valid in such expressions.

CHAPTER 4: RESERVED WORDS

ABS	IF	REM
AND	INPUT	RESTORE
ASC	INT	RETURN
ATTACH	LEFT\$	RIGHT\$
BYE	LEN	RND
CALL	LET	RUN
CHR\$	LIST	SGN
CLEAR	LOG	SQR
CONT	MID\$	SPC
DATA	NEW	STEP
DEF	NEXT	STOP
DIM	NOT	STR\$
END	ON	TAB
EXP	OR	THEN
FN	PEEK	TO
FOR	POKE	TROFF
FRE	POS	TRON
GET	PRINT	VAL
GOSUB	?	
GOTO	READ	

CHAPTER 5: LINE NUMBERS, NUMBERS, AND STRINGS

LINES

Maximum length: 80 characters (after Basic has edited a line into storable form)

Minimum value: 0

Maximum value: 63999

Values allowed: any integer value from minimum to maximum.

NUMERIC VALUES

Maximum/minimum value: approximately $\pm 1.70141 \times 10^{38}$

Tiniest nonzero value: approximately $\pm 2.93874 \times 10^{-39}$

Maximum precision: approximately 9 decimal digits

INTEGER VALUES

The limits on integer values apply only to the values of integer *variables*. A numeric variable may have an "integer value" (a value that contains no decimal part) in the range $\pm 999,999,999$.

Maximum value: 32767

Minimum value: -32768

RULES FOR DISPLAYING NUMERIC VALUES (PRINT AND STR\$)

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 number is an integer (with no decimal part) in the range $\pm 999,999,999$, it is displayed as an integer, with no decimal point.

Examples: -32 0 325000

3. If the absolute value of the number is in the range $.01 \leq n < 1,000,000,000$ the number is displayed as a real value (with a decimal point).

Examples: .1 -1 3.141592
 9111.11111 -9111.11111

4. 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 $1 \leq n < 10$. If it can be expressed as an integer in that range, it is displayed as an integer. Otherwise it is displayed as a real number.

Examples: 3E + 14 3.14E + 14

'E' is followed by a sign, '+' or '-', and a two-digit exponent.

Examples: 3.14E + 14 3E - 03

RULES FOR READING NUMERIC 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 an integer.

1. If a number is negative, the sign must be '-'. Otherwise the sign may be '+' or may be absent. Leading blanks are not allowed.

2. After the sign, if any, is an integer constant or numeric constant.

Basic 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', Basic reads '50'. If the number is 'b50' (with a leading blank), Basic reads '0', since leading

blanks are not allowed.

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 must be in upper 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

CHAPTER 6: BASIC PROGRAM EDITOR QUICK REFERENCE

ADDING, DELETING, REPLACING, OR COPYING A LINE

<u>Change To Be Made</u>	<u>Procedure</u>
--------------------------	------------------

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. <i>or</i> (2) LIST a line and delete everything except the line number, then press ENTER to make Basic accept the edited line.
Replace a line	Type a new line with the same line number as the line you want to replace. <i>or</i> LIST a line and type over everything except the line number.
Copy a line	LIST a 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.

CURSOR MOTION

<u>Editing Operation</u>	<u>Keystrokes</u>
Move cursor left	←
Move cursor right	→
Move cursor left to end of line	LOCK ←
Move cursor right to end of line	LOCK →

INSERT CHARACTERS

<u>Editing Operation</u>	<u>Keystrokes</u>
Insert a character, <i>x</i>	INSERT <i>x</i>
Insert multiple characters	LOCK INSERT <i>xxxxx...</i> Press INSERT again when done.
Insert a space; move cursor right	INSERT ▶
Insert a space; don't move cursor	INSERT ◀
Insert multiple spaces; move cursor right	LOCK INSERT ▶▶▶ . . . Press INSERT again when done.
Insert multiple spaces; don't move cursor	LOCK INSERT ◀◀◀ . . . Press INSERT again when done.
Insert spaces continuously; move cursor right	LOCK INSERT LOCK ▶ Press INSERT again when done.
Insert spaces continuously; don't move cursor	LOCK INSERT LOCK ◀ Press INSERT again when done.

DELETE CHARACTERS

<u>Editing Operation</u>	<u>Keystrokes</u>
Delete character under cursor; don't move cursor	DELETE ▶
Delete character under cursor; move cursor left	DELETE ◀
Delete characters from cursor right; don't move cursor	LOCK DELETE ▶▶▶ . . . Press DELETE again when done.
Delete characters from cursor left; move cursor left	LOCK DELETE ◀◀◀ . . . Press DELETE again when done.
Delete characters from cursor to end of line	LOCK DELETE LOCK ▶ Press DELETE again when done.
Delete characters from cursor to start of line	LOCK DELETE LOCK ◀ Press DELETE again when done.

SHORTCUTS

<u>Editing Operation</u>	<u>Keystrokes</u>
Review a line longer than LCD	ROTATE Press ROTATE again 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.

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

<u>numeric value</u>	<u>name</u>	<u>ASCII std</u>	<u>typical meaning</u>
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.
128	cursor up	no	Moves cursor up one line.
129	cursor left	no	Moves cursor left one character; does not erase character cursor is moved to.

<u>numeric value</u>	<u>name</u>	<u>ASCII std</u>	<u>typical meaning</u>
130	cursor right	no	Moves cursor right one character; does not replace character under cursor with a blank.
131	cursor down	no	Moves cursor down one line (equivalent to line feed).

ESCAPE CONTROL SEQUENCES

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

<i>byte</i>	<i>meaning</i>
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. Unlike peripherals, it has no device code. Thus it cannot be attached to another LUN with the ATTACH statement. You can attach it to a device with a POKE into the SDT. This POKE and warnings about its use are described in Chapter 8.

Technical Information

ATTACH device code: none; see note above.

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. Like the keyboard, it has no device code. Thus it cannot be attached to another LUN with the ATTACH statement. You can attach it to a device with a POKE into the SDT. This POKE is described in Chapter 8.

Technical Information

ATTACH device code: none. See note above.

Control codes:

<u>Code</u>	<u>Meaning</u>	<u>Function on device</u>
7	Bell	Makes the HHC beep.
8	Backspace	Backspaces the cursor, erasing the character the cursor was previously at. ¹¹¹

<u>Code</u>	<u>Meaning</u>	<u>Function on device</u>
12	Form feed	A no-operation on the LCD.
13	Carriage rtn	Clears display and moves cursor to left edge of LCD.
27	Escape	Begins an escape control sequence.
129	Cursor left	Non-destructive backspace. ^{1}
130	Cursor right	Non-destructive space.

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.

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 another 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 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.

^{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: 68

Control codes:

<u>Code</u>	<u>Meaning</u>	<u>Function on device</u>
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 ADAPTOR

The serial interface adaptor 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 Institute of Electrical and Electronic Engineers (IEEE), 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 interface program's configuration 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.

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 interface's configuration 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, see the manual that accompanies the serial interface's program capsule for instructions.

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.

Note About Protocols

The serial interface'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. If neither is used, 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.

⁽²⁾ - an invisible file is one that does not appear in the file system editor's menu or in Basic's menu.

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 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 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.

<u>The byte at:</u>	<u>represents LUN:</u>
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:

<u>value</u>	<u>means</u>
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.

To attach the keyboard or LCD to a LUN, POKE 0 or 6 into the proper SDT entry. (The ATTACH statement does not work for these devices, since they have no device codes.)

Note: never leave a device attached to more than one LUN

at a time! If you do so, your program may behave unpredictably.

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.

Note: do not press any key on the HHC's keyboard when LUN #0 is attached to a peripheral device. If you do so, the HHC may "freeze up" and execute only one Basic statement each time you press a keyboard key. If this happens, the HHC will remain "frozen" until you return to the menu or reattach LUN #0 to the keyboard.

Note: do not "type ahead" with the HHC keyboard or any peripheral attached to LUN #0.

The ATTACH Statement

The ATTACH statement does not give error messages, even when it fails. In the normal course of things, your first indication of an unsuccessful ATTACH is an IO error when you try to do an INPUT, GET, or PRINT.

To determine whether the most recent ATTACH succeeded, PEEK at location 926:

<u>value</u>	<u>means</u>
0	most recent ATTACH failed, or no ATTACH done yet.
1	most recent ATTACH succeeded

THE HELP AND I/O KEYS

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

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

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.

Note: while you are in Basic, the I/O key may only be used to determine how much free space remains in a memory area other than the current memory area, and to determine whether a peripheral is on or off.

To determine how much free space remains in the current memory area, use the FRE function. To turn a peripheral on or off, return to the Basic menu or the primary menu before using the I/O key.

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.

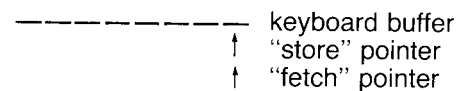
Note: you must be very careful because one character can be removed from the buffer for each Basic instruction executed.

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:



⁽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 Basic from the primary menu. The discussion of the process is accurate, however.

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.

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

<u>location</u>	<u>contains</u>
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$="":PC=PEEK (522) buffer.
1130 REM Accumulate characters from
1140 PC=PC-1:IF PC<0 THEN RETURN
1150 PK$=PK$+CHR$(PEEK (628+PC)):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 buffer 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:

<u>location</u>	<u>contents</u>
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:

<u>to “type” a function key</u>	<u>perform POKES</u>
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:

<u>value</u>	<u>STOP/SPEED setting</u>
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

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:

<u>location</u>	<u>counts units of</u>
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
```

After performing this subtraction, the result should represent a value small enough to hold in a numeric variable without loss of precision:

```
850 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:

<u>value</u>	<u>rotation mode</u>
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.

VARIABLES AND ARRAYS

Locations of Variables and Arrays

For some kinds of mathematical calculations it is useful to be able to PEEK and POKE the parts of a numeric variable directly.

For example, if you must multiply two numbers together and first want to determine whether the product will overflow, you can PEEK at the two numbers' exponents, add them, and determine whether the sum is over or near the maximum. (See the definition of the trigonometric function ATAN2 in Chapter 2 for an example of this.)

Basic stores all variables consecutively in memory. It stores all arrays consecutively in another part of memory.

The locations that point to the variable and array areas are:

<u>location</u>	<u>contents</u> ⁽²⁾
200-201	Address of beginning of variable area. Note , this address changes when you edit your stored program.
202-203	Address of end of variable area and beginning of array area. Note , this address changes when you edit your stored program or use a new variable.
204-205	Address of end of array area. Note , this address changes when you edit your stored program, or use a new variable or array.

⁽²⁾ - In each two-byte integer value or address, the first byte is the least significant and the second byte is the most significant. Thus, you can reconstruct the address of the beginning of the variable area (for example) like this:

```
AD=PEEK(200)+256*PEEK(201)
```

Formats of Variables

Variables are stored in the variable area in the order that the program first refers to them. Each variable occupies 7 bytes.

A numeric variable has the following format:

<u>byte</u>	<u>contents</u>
0	First character of the variable name, in ASCII.
1	Second character of the variable name, in negative ASCII. ⁽³⁾ If the name is one character long, PEEK(byte 1) = 0.
2-6	Value of the variable in numeric format (see the Format of a Numeric Value section in the following text.)

An integer variable has the following format:

<u>byte</u>	<u>contents</u>
0	First character of the variable name, in negative ASCII.
1	Second character of the variable name, in negative ASCII. If the name is one character long, PEEK(byte 1) = 128.
2-3	Value of the variable in two's complement binary form. ⁽²⁾
4-6	Unused.

A string variable has the following format:

<u>byte</u>	<u>contents</u>
0	First character of the variable name, in negative ASCII.
1	Second character of the variable name, in ASCII. If the name is one character long, PEEK(byte 1) = 0.
2	Length in characters of the string value.

⁽³⁾ - To convert negative ASCII to ordinary ASCII, subtract 128 from the value returned by PEEK.

- 3-4 Address of the first character of the string value. The characters in the value are stored consecutively. **Note**, this address may change any time you create a new string variable or array, or change the value of an existing one.
- 5-6 Unused.

Formats of Arrays

Arrays are stored in the array area in the order that the program first refers to them.

All arrays have the following format:

<u>byte</u>	<u>contents</u>
0-1	Array name, in ASCII and/or negative ASCII. The conventions used to indicate the type of the array are the same as for variables.
2-3	Length of this array, including the name, length field, etc. Add this length to the address of this array to get the address of the next array (if any).
4	Number of dimensions.
5...	Size of each dimension. Each size field is two bytes long. The sizes of the dimensions are given in reverse order, <i>i.e.</i> , the rightmost dimension is given first, the leftmost last.
varies	After the dimensions come the elements. In multi-dimensional arrays, the first element varies fastest. For example, in an array XY dimensioned (1,1), the elements would be stored in the order (0,0), (1,0), (0,1), (1,1).

In a numeric array, each element is five bytes long, and is stored in variable format (see below).

In an integer array, each element is two bytes long, and is stored in two's complement binary notation.

In a string array, each element is three bytes long. The first byte gives the length in characters of the value of the element. The second and third bytes contain the address of

the first character of the element. **Note**, this address may change any time you create a new string variable or array, or change the value of an existing one.

Format Of a Numeric Value

If a numeric value is expressed as

$$n = s \cdot m \cdot 2^e$$

where

s = the sign, ± 1 ,

m = the mantissa, in the range $1.0 \leq m < 2.0$,

e = the exponent, in the range ± 127 ,

then the variable's format is:

<u>bytes(bits)</u>	<u>contents</u>
0(0)-0(7)	$128 + e$ in binary form. If this field is 0, then the numeric value is 0, even if $m=0$.
1(0)	Sign bit. If $s = -1$ ($n < 0$), the bit is on. If $s = 1$ ($n \geq 0$), the bit is off.
1(1)-4(7)	$m - 1.0$. Stored as a 31-bit binary fraction with the "binary point" before the first bit.

Other Useful Addresses

Here are some other useful addresses relating to variables and programs:

<u>location</u>	<u>contents</u>
71-72	Address of the variable or array element most recently referred to. If the variable or array is numeric or integer, this is the address of the value. If it is string, this is the address of the length byte. Note : in an assignment statement of the form $XX = \text{PEEK}(71) + 256 * \text{PEEK}(72)$, the "variable most recently referred to" is XX!
212-213	Line number of the line currently being executed. In immediate mode, the "line

number" is greater than 63,999 (the largest valid value).

- 214-215 Line number of the line that was being executed when the most recent of the following events occurred in deferred mode:
1. Execution of a STOP statement.
 2. Execution of an END statement.
 3. Program interrupted by the C1 key.
- 218-219 Line number of the DATA statement currently being read.

Finding a Variable's Value

The most convenient way to find a variable's value is to refer to the variable, then PEEK at location 71-72 to get the address of the value. For example, the following code may be used to assign EX the value of the exponent of VL:

```
500 EX=VL
510 EX=PEEK (71) + 256*PEEK (72)
520 EX=PEEK (EX)
```

Note that this statement would *not* work if line 500 said 'VL=VL', since line 510 assigns EX the address of itself.

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\$(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'.

<u>numeric value</u>	<u>HHC name</u>	<u>ASCII name</u>
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

<u>numeric value</u>	<u>HHC name</u>	<u>ASCII name</u>
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

<u>numeric value</u>	<u>HHC key</u>	<u>HHC graphic</u>	<u>LCD action</u>
0			
1			
2			
3			
4			
5			
6			
7	ROTATE		"beep"
8			Backspace cursor; erase character under cursor after backspace.
9			
10			
11	IO		
12			
13	ENTER		Erase LCD; move cursor to left edge.
14	STOP SPEED ⁽¹⁾		
15			
16			
17			
18			
19			

⁽¹⁾. This key has an immediate function in Basic, and so cannot normally be read by GET. Note that *none* of the control characters #0 through #31 can be read by INPUT.

<u>numeric value</u>	<u>HHC key</u>	<u>HHC graphic</u>	<u>LCD action</u>
20	HELP		
21	f1 ⁽¹⁾		
22	f2 ⁽¹⁾		
23	f3 ⁽¹⁾		
24			
25			
26			
27			Begins an escape control sequence.
28			
29			
30			
31			

DISPLAYABLE CHARACTERS

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

<u>numeric value</u>	<u>HHC keybd character</u>	<u>name</u>
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

<u>numeric value</u>	<u>HHC keybd character</u>	<u>name</u>
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	`	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	

<u>numeric value</u>	<u>HHC keybd character</u>	<u>name</u>
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.

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

<u>numeric value</u>	<u>HHC name</u>	<u>HHC key</u>	<u>HHC graphic</u>	<u>LCD action</u>
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⁾	⊞	
133	"PM" symbol	DELETE ⁽³⁾	⊞	
134	superscript M	⁽⁴⁾	⊞	
135	division sign	⁽⁴⁾	÷	

⁽²⁾ - INPUT reads this character as ENTER.

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

⁽⁴⁾ - INPUT reads this character as SPACE.

<u>numeric value</u>	<u>HHC name</u>	<u>HHC key</u>	<u>HHC graphic</u>	<u>LCD action</u>
136	"times" sign	(4)	×	
137	block cursor	SEARCH ⁽²⁾	█	
138	"delete" cursor		□	
139	'a' with umlaut	C1 ⁽³⁾	ä	Except when read by GET, causes "break" in execution of the program that is running.
140	'o' with umlaut	C2	ö	
141	'u' with umlaut	C3	ü	
142	'n' with tilde	C4	ñ	

CHAPTER 10: ERRORS

ERROR MESSAGE FORMAT

When Basic detects an error in immediate mode, it beeps and displays a message saying

```
xx ERROR
```

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

When Basic detects an error in deferred mode, it beeps and displays a message saying

```
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.

ERROR CODES

BS - Bad Subscript

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.

CN - Can't Continue

You tried to CONTINUE a program when you have not done a RUN, after execution was interrupted by an error, or after you edited the program.

DD -- Duplicate Dimension

You tried to define an array with DIM after the array was already defined. This error often occurs because the array was defined implicitly (by using one of its elements) before the DIM statement was executed.

FC -- Function Call

You gave a function an argument whose value was out of range. Check the description of the function in question for the permissible range of values.

ID -- Illegal Direct

You tried to execute a statement in immediate mode that is only allowed in deferred mode.

IO -- I/O Error

First possible cause: you tried to do an I/O operation on a LUN that was not successfully ATTACHED to a peripheral device. Note that an unsuccessful ATTACH does not produce this error; the first I/O operation on the LUN produces it.

Second possible cause: an I/O operation failed to complete successfully. An I/O operation may fail to complete because the operation is illegal (*e.g.*, an INPUT operation on a LUN attached for output), or because the device is improperly set up or out of order.

This error can occur if you try to do an I/O operation on a LUN that has been "unattached" by a POKE.

LS -- Length of String

You tried to use the concatenation operation, '+', to create a string more than 255 characters long.

NF -- 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 -- Out of Data

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

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 and arrays; the program has too many nested FOR/NEXT loops; the program has too many

nested GOSUBs; or an expression is too complicated.

OV -- Overflow

You tried to perform a calculation whose result was too large to be represented in Basic's numeric format. The largest number Basic can represent is approximately 1.70141×10^{38} .

Note that underflow does *not* produce an error in Microsoft Basic. It just returns a zero result.

RG -- 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.

RT -- Run Time Error

This message indicates an error in an operation which Basic requested one of the HHC's intrinsic programs to perform. It is often associated with an invalid I/O operation.

SN -- Syntax Error

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, etc.

ST -- String Formula Too Complex

You tried to use a string expression too complex for Basic to process. Break it up into two or more shorter expressions.

TM -- Type Mismatch

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.

UF -- Undefined Function

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

US -- Undefined Statement

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

/0 -- Division by Zero

You tried to divide a number by zero. Note that dividing zero by zero produces this error.

INDEX

A

Array

Format of, 8-12
Last referred to, 8-13
Location of, 8-10

ASCII, 7-1

Character set, 9-3
DELETE character, 9-5

Assignment statement, 1-2

ATTACH

POKE and, 8-1
ATTACH statement, 1-3
Errors in, 8-2

B

Body of function definition, 1-6

BYE statement, 1-3

C

C1 key, 1-4

Line number of line interrupted by, 8-14

CALL statement, 1-4

Character

Displayable character set, 9-3

CLEAR key, 1-3

CLEAR statement, 1-4

Clock unit, 8-8

CONT statement, 1-4, 10-1

Control character, 7-1, 9-1

Control ROM, 7-10

Cursor, 7-1

D

DATA statement, 1-4, 1-5, 1-20, 10-2

Date, 8-8

DEF statement, 1-6

DIM statement, 1-7, 10-1

E

END statement, 1-4, 1-7

Line number of most recent, 8-14

ENTER key, 1-10

Error

Message format, 10-1

Escape control sequence, 7-2, 9-1

ETX/ACK protocol, 7-9, 7-11

F

FOR statement, 1-4, 1-8, 1-15, 1-16, 10-2
FOR/NEXT statement, 10-2
Formal parameter, 1-6
Function
 DEF statement, 1-6
Function key, 1-10
 Defining a, 8-6
 Precedence over keyboard & pushkey buffers, 8-7
 Simulating keystrokes, 8-6
Functions
 Trigonometric, 2-4

G

GET statement, 1-10
GOSUB statement, 1-4, 1-11, 1-20, 10-2, 10-3
GOTO statement, 1-11
Graphics, 7-6

H

HELP key
 Using in Basic, 8-2

I

I/O
 Error, 10-2
I/O key
 Using in Basic, 8-2
IEEE, 7-7
IF statement, 1-12
INPUT statement, 1-12
Institute of Electrical and Electronic Engineers, **see** IEEE
Integer value
 Range of, 5-1

K

Key
 2nd SFT, 1-10
 C1, 1-4
 CLEAR, 1-3
 ENTER, 1-10
 Function, 1-10, 8-6
 HELP, 8-2
 I/O, 8-2
 SHIFT, 1-10
 STOP/SPEED, 8-7

Keyboard, 7-5

 Additional characters, 9-5
 Control character input, 9-1
 Displayable character set, 9-3
 Keyboard buffer, 8-3
 Reattaching, 8-2

L

LCD, 7-5

 Additional characters, 9-5
 Control character display, 9-1
 Displayable character set, 9-3
 Reattaching, 8-2
 Rotation mode control, 8-9
 STOP/SPEED control, 8-7
LET statement, 1-2, 1-13
Line
 Maximum length of, 5-1
Line number
 Currently being executed, 8-13
 Most recent STOP, END, or break, 8-14
 Range of, 5-1
LIST statement, 1-14
Logical expression, 1-12
LUN
 Attaching keyboard or LCD to, 8-1

M

Machine language subroutine, 1-4
Message (ETX/ACK protocol), 7-9
Micro printer, 7-6
Modem, 7-10

N

NEW statement, 1-15
NEXT statement, 1-4, 1-8, 1-15, 10-2
Number Overflow, 10-3
Numeric value
 Format of, 8-13
 Range of, 5-1
 Rules for display, 5-1
 Rules for input, 5-2

O

ON/GOSUB statement, 1-16
ON/GOTO statement, 1-16
Overflow, 10-3

P

POKE statement, 1-16
PRINT statement, 1-16
Protocol, 7-8
 ETX/ACK, 7-9
 XON/XOFF, 7-8
Pushkey buffer, 8-4
 Precedence over keyboard buffer, 8-4

R

READ statement, 1-5, 1-18, 1-20, 10-2
REM statement, 1-19
RESTORE statement, 1-4, 1-20
RETURN statement, 1-11, 1-20, 10-3
ROM
 Control, 7-10
Rotation mode, 8-9
RS-232 interface, 7-7

S

SDT

 Addresses in, 8-1
Serial interface, 7-7
SHIFT key, 1-10

Statement

 Assignment, 1-2
 ATTACH, 1-3, 8-2
 BYE, 1-3
 CALL, 1-4
 CLEAR, 1-4
 CONT, 1-4, 10-1
 DATA, 1-4, 1-5, 1-20, 10-2
 DEF, 1-6
 DIM, 1-7, 10-1
 END, 1-4, 1-7
 FOR, 1-4, 1-8, 1-15, 1-16, 10-2
 FOR/NEXT, 10-2
 GET, 1-10
 GOSUB, 1-4, 1-11, 1-20, 10-2, 10-3
 GOTO, 1-11

IF, 1-12
INPUT, 1-12
LET, 1-2, 1-13
LIST, 1-14
NEW, 1-15
NEXT, 1-4, 1-8, 1-15, 10-2
ON/GOSUB, 1-16
ON/GOTO, 1-16
POKE, 1-16
PRINT, 1-16
READ, 1-5, 1-18, 1-20, 10-2
REM, 1-19
RESTORE, 1-4, 1-20
RETURN, 1-11, 1-20, 10-3
STOP, 1-4, 1-20
TROFF, 1-21
TRON, 1-21

STOP statement, 1-4, 1-20
 Line number of most recent, 8-14
STOP/SPEED key, 8-7
String, 10-3
 DATA statement, 1-5
 Length, 10-2
Subroutine
 Machine language, 1-4

T

Telecomputing system, 7-11
Television, 7-6
Time, 8-8
Trace, 1-21
Trigonometric functions, 2-4
TROFF statement, 1-21
TRON statement, 1-21
TV Adaptor, 7-6

V

Variable

 Format of, 8-11
 Last referred to, 8-13
 Location of, 8-10

W

Word break character, 7-4
Word-wrap, 7-4

X

XON/XOFF protocol, 7-8, 7-11

Z

Zone, 1-17



FRIENDS AMIS, INC.

The program described in this document is furnished under a license and may be used, copied, and disclosed only in accordance with the terms of such license.

Friends Amis, Inc. ("FA") EXPRESSLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR USE FOR A PARTICULAR PURPOSE RESPECTING THE HHC SOFTWARE PROGRAM AND MANUAL. THE PROGRAM AND MANUAL ARE SOLD "AS IS". THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR USE FOR A PARTICULAR PURPOSE AS TO THE MEDIUM ON WHICH THE SOFTWARE IS RECORDED ARE LIMITED TO SIXTY (60) DAYS FROM THE DATE OF LICENSING BY THE INITIAL USER OF THE PRODUCT AND ARE NOT EXTENDED TO ANY OTHER PARTY.

USER AGREES THAT ANY LIABILITY OF FA HEREUNDER, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE LICENSE FEE PAID BY USER TO FA. FA SHALL NOT BE LIABLE FOR INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES, SUCH AS, BUT NOT LIMITED TO, LOSS OR INJURY TO BUSINESS, PROFITS, GOODWILL, OR FOR EXEMPLARY DAMAGES, EVEN IF FA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

FA will not honor any warranty when the product has been subjected to physical abuse or used in defective or non-compatible equipment.

The user shall be solely responsible for determining the appropriate use to be made of the program and establishing the limitations of the program in the user's own operation.

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.