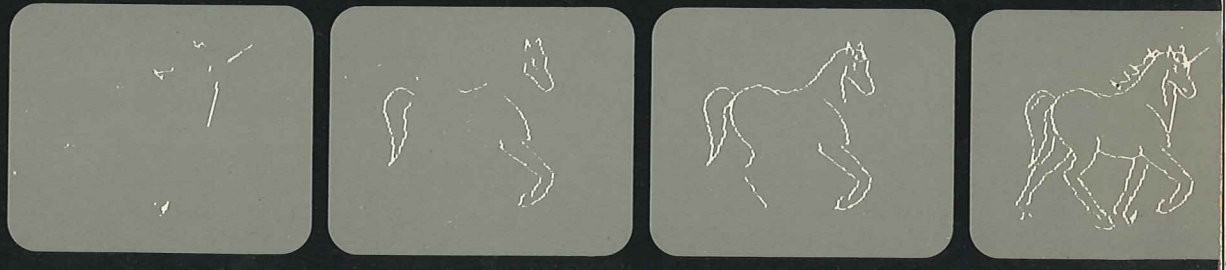
GIGI™

GIGI BASIC Manual



digital

DIGITAL EQUIPMENT CORPORATION

January 1981

GIGI BASIC Manual

Graham REES.

Order No. AA-K335A-TK

This manual provides information on how to run GIGI BASIC; also it provides descriptions of the features of the GIGI BASIC language.

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

DIGITAL EQUIPMENT CORPORATION

First Printing, January, 1981

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

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

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

This document includes material from Microsoft BASIC manuals. Copyright© 1980 by Microsoft. All rights reserved.

Copyright© January 1981 Digital Equipment Corporation. All rights reserved.

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	FOCAL
DECnet	IAS
DECsystem-10	PDP
DECtape	RSX
DECUS	UNIBUS
DIBOL	VAX
DIGITAL	VMS

CONTENTS

PREFACE

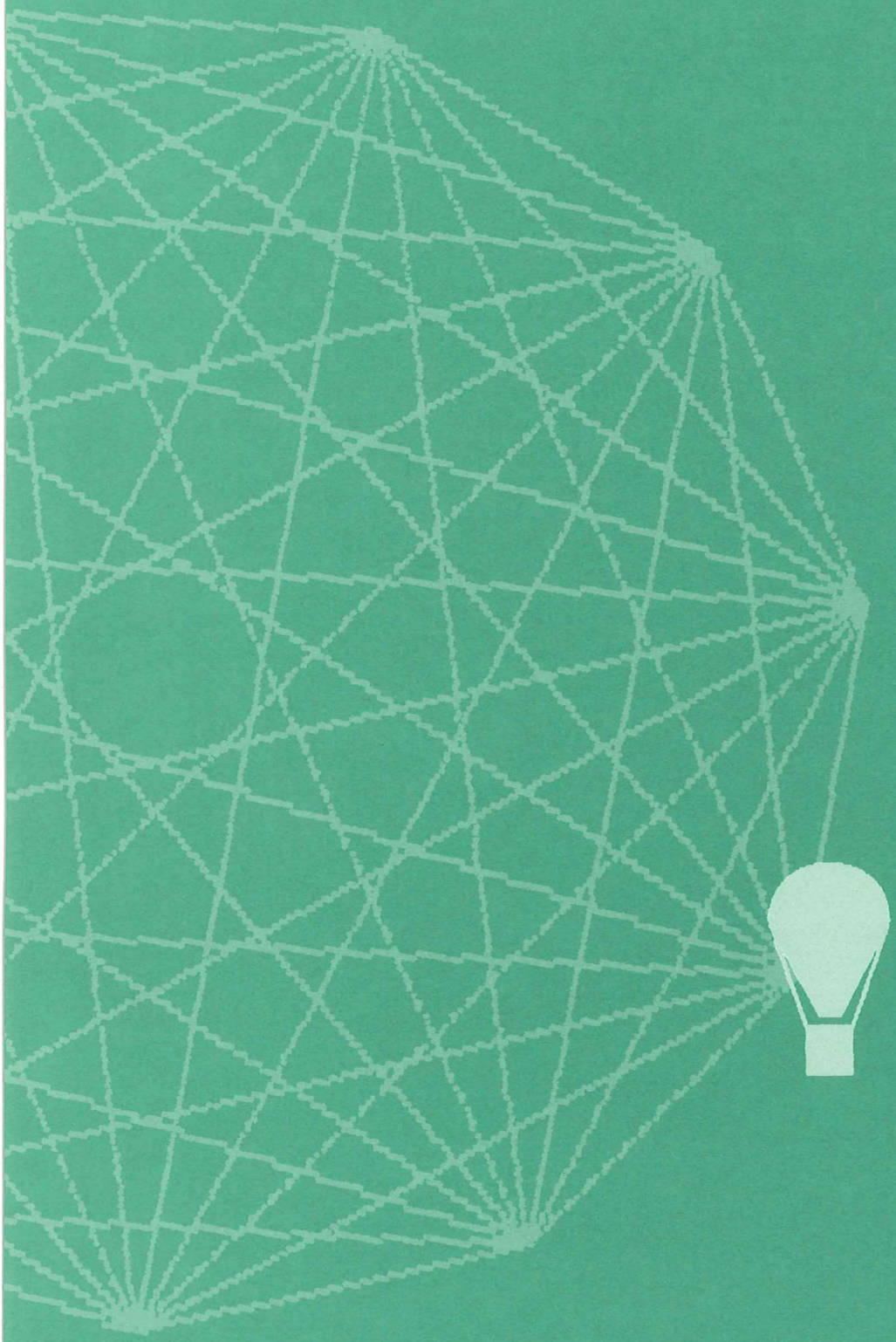
The manual is divided into three large chapters plus a number of appendices. Chapter 1 covers a variety of topics, largely pertaining to information representation when using GIGI BASIC. Chapter 2 contains the syntax and semantics of every command and statement in GIGI BASIC, ordered alphabetically. Chapter 3 describes all of GIGI BASIC's intrinsic functions, also ordered alphabetically. The appendices contain lists of error messages, ASCII codes, and mathmatical functions.

This manual is a companion to the *GIGI/ReGIS Handbook*, Order Number AA-K336A-TK.The *GIGI ReGIS/Handbook* should be consulted concerning normal terminal operation and graphics programming.

GIGI BASIC does not implement certain BASIC language functions in the same way that DEC BASIC implements these functions. Where GIGI BASIC and DEC BASIC differ in implementation, those differences are indicated by grey type, as shown in this paragraph.

1

General Information About GIGI BASIC



1 GENERAL INFORMATION ABOUT GIGI BASIC

This BASIC is a tailored version of Microsoft BASIC. It is specifically designed for the GIGI terminal. BASIC is provided as a tool to be used in making GIGI an intelligent terminal. For this reason, only a limited amount of user memory is provided with GIGI. It is therefore recommended that applications and instructional programming be done on the host computer as there is no guarantee of source language transportability between the BASIC in this version of GIGI and that provided in the future.

LOCAL BASIC VS. HOST BASIC

GIGI BASIC can be run in either of two modes. One mode is called local BASIC. In local BASIC mode, the terminal user is in control of the GIGI BASIC system; it will take commands and programs from the user, and program input and output default to the keyboard and display. The other mode is host BASIC. In host BASIC mode, the host computer is in control of the GIGI BASIC system; commands and programs come from the host computer, and all input and output default to the host computer.

Most likely the terminal user will want to be in local BASIC mode. Host BASIC mode is normally used only when a host program needs to operate or control the GIGI BASIC system.

STARTING GIGI BASIC

GIGI BASIC can be started either by entering SET-UP mode from the keyboard or by transmission of one of the appropriate control sequences from the host computer.

From SET-UP

Press the SET-UP key to enter SET-UP mode on the terminal (consult the *GIGI/ReGIS Handbook*, if needed). Then type BA1 to enter local BASIC mode, or BA2 to enter host BASIC mode. Pressing SET-UP again will leave SET-UP mode and start GIGI BASIC.

From a host program

The SET-UP mode DCS sequence can be sent from the host to directly specify BA1 (for local BASIC) or BA2 (for host BASIC). The Set Mode ANSI control sequence can also be used to select either local or host BASIC; they have the additional effect of terminating any running BASIC program, whereas the other methods will resume a program that may have been running before. (See the *GIGI/ReGIS Handbook* for more information on control sequences.)

GIGI BASIC can be suspended and GIGI returned to normal terminal operation, either by setting SET-UP BA0 (from either SET-UP mode or the host), or by sending either Reset Mode ANSI sequence corresponding to the Set Mode sequences.

MODES OF OPERATION

When GIGI BASIC is initialized, it types the prompt "Ok." "Ok" means GIGI BASIC is at command level, that is, it is ready to accept commands. At this point, GIGI BASIC may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a calculator for quick computations that do not require a complete program.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

LINE FORMAT

Program lines in a BASIC program have the following format (square brackets indicate optional):

nnnnn BASIC statement[:BASIC statement . . .] *carriage return*

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of 255 characters.

Line Numbers

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65529. A period (.) may be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

CHARACTER SET

The GIGI BASIC character set is comprised of alphabetic characters, numeric characters and special characters.

The alphabetic characters in GIGI BASIC are the upper case and lower case letters of the alphabet.

The numeric characters in GIGI BASIC are the digits 0 through 9.

The following special characters and terminal keys are recognized by GIGI BASIC:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
–	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
@	At-sign
_	Underscore
<DEL>	Deletes last character typed.
<ESC>	Escapes Edit Mode Insert command.
<TAB>	Moves print position to next tab stop. Tab stops are every eight columns.
<LF>	Moves to next physical line.
<CR>	Terminates input of a line.

Control Characters

The following control characters are in GIGI BASIC:

<CTRL>C	Interrupts program execution and returns to GIGI BASIC command level.
<CTRL>G	Rings the bell at the terminal.
<CTRL>O	Halts program output while execution continues. A second <CTRL>O restarts output.
<CTRL>R	Retypes the line that is currently being typed.
<CTRL>U	Deletes the line that is currently being typed.

CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks.

Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

Integer Constants: Whole numbers between −32768 and +32767. Integer constants do not have decimal points.

Fixed Point Constants: Positive or negative real numbers, i.e., numbers that contain decimal points.

Floating Point Constants: Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10−38 to 10+38.
Examples:

```
235.988E-7 = .0000235988
2359E6 = 2359000000
```

Hex Constants: Hexadecimal numbers with the prefix &H.

Examples:

```
&H76
&H32F
```

Octal constants: Octal numbers with the prefix &O or &.

Examples:

```
&O347
&1234
```

Although there are five forms of external constants, all internal numeric values in GIGI BASIC are floating point constants with 24 bits of precision.

VARIABLES

Variables are names used to represent values that are used in a BASIC program.

The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

Variable Names and Declaration Characters

GIGI BASIC variable names may be any length, however, only the first two characters are significant. The characters allowed in a variable name are letters and numbers. The first character must be a letter.

A variable name may not be a reserved word. A reserved word may not be embedded in a variable name. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all GIGI BASIC commands, statements, function names and operator names.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character, that is, it declares that the variable will represent a string.

Examples of GIGI BASIC variable names:

N$	declares a string value
ABC	represents a single precision value

Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with integers or integer expressions. An array variable name has as many subscripts as there are dimensions in the array.

For example, V(10) would reference a value in a one-dimensional array, T(1,4) would reference a value in a two-dimensional array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by GIGI BASIC may be divided into four categories:

- Arithmetic
- Relational
- Logical
- Functional

Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
–	Negation	–X
*,/	Multiplication, Floating Point Division	X*Y X/Y
+,–	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression
X+2Y	X+Y*2
$X-\frac{Y}{Z}$	X–Y/Z
	X*Y/Z
	(X+Y)/Z
(X	(X^2)^Y
X	X^(Y^Z)
X(–Y)	X*(–Y) Two consecutive operators must be separated by parentheses.

Overflow and Division by Zero

If, during the evaluation of an expression, a division by zero is encountered, the "?/0" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "?/0" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "?OV" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

Relational Operators

Relational operators are used to compare two values. The result of the comparison is either true (−1) or false (0).This result may then used to make a decision regarding program flow. (See IF, Chapter 2.)

Operator	Relation Tested	Expression
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See LET, Chapter 2.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

X+Y < (T−1)/Z

is true if the value of X plus Y is less than the value of T−1 divided by Z. More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either true (not zero) or false (zero).

In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF, Chapter 2). For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range −32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or −1, logical operators return 0 or −1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to mask all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to merge two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
−1 AND 8=8	−1 = binary 1111111111111111 and 8 = binary 1000, so −1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
−1 OR −2=−1	−1 = binary 1111111111111111 and −2 = binary 1111111111111110, so −1 OR −2 = −1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of −1.
NOT X=−(X+1)	The two's complement of any integer is the bit complement plus one.

Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. GIGI BASIC has intrinsic functions that reside in the system, such as SQR (square root) or SIN (sine). All of GIGI BASIC's intrinsic functions are described in Chapter 3.

GIGI BASIC also allows user-defined functions that are written by the programmer.

See DEF FN, Chapter 2.

String Operations

Strings may be concatenated using + . For example:

```
10      A$ = " FILE"   :  B$ = " NAME"
20      PRINT  A$   +   B$
30      PRINT   " NEW   "   +   A$   +   B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=       < >       < >       < =       > =
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
" AA" < " AB"
" FILENAME" = " FILENAME"
" X&" > " X#"
" CL" > " CL "
" kg" > " KG"
" SMYTH" < " SMYTHE"
B$   <   " 9/12/78"      where   B$   =   8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the DELETE key. The DELETE key has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type < CTRL > U. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. GIGI BASIC will automatically replace the old line with the new line.

See EDIT, Chapter 2.

To delete the entire program that is currently residing in memory, enter the NEW command. (See Chapter 2.)NEW is usually used to clear memory prior to entering a new program.

ERROR MESSAGES

If GIGI BASIC detects an error that causes program execution to terminate, an error code is printed. For a complete list of GIGI BASIC error codes and their meanings, see Appendix A.

GRAPHICS CONTROL

GIGI BASIC causes the terminal to leave graphics mode and return to normal text terminal operation when any of the following occur:

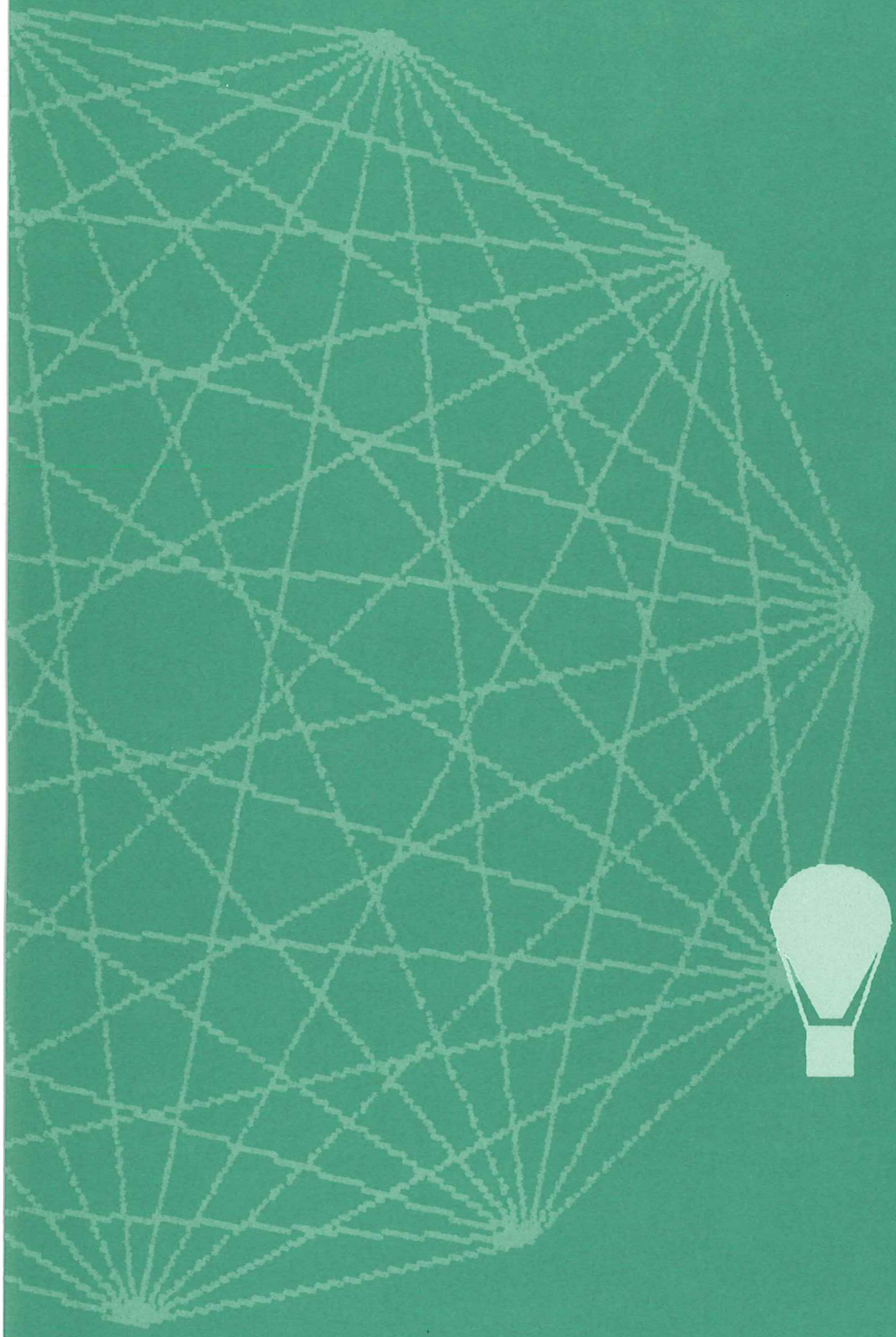
- a BASIC program terminates due to an END or STOP statement.
- an error message is displayed.
- a <CTRL>C stop is detected.
- the "Ok" message is displayed.

This prevents error messages from being lost if they are issued while the terminal is in graphics mode. It also means that it is not necessary to turn off graphics mode at the end of a graphics program, although it would be a poor programming practice to rely on this since other programming systems do not automatically turn off graphics.

GIGI BASIC provides built-in control strings to allow the BASIC program to enter and exit graphics mode. See the GON$ and GOFF$ functions in Chapter 3.

2

GIGI BASIC Commands and Statements



2 GIGI BASIC COMMANDS AND STATEMENTS

All of the GIGI BASIC commands and statements are described in this chapter. Each description is formatted as follows:

Format: Shows the correct format for the instruction. See below for format notation.

Purpose: Tells what the instruction is used for.

Remarks: Describes in detail how the instruction is used.

Example: Shows sample programs or program segments that demonstrate the use of the instruction.

Format Notation

Wherever the format for a statement or command is given, the following rules apply:

- Items in capital letters must be input as shown.
- Items in lower case italic letters are to be supplied by the user.
- Items in square brackets ([ ]) are optional.
- All punctuation except square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.
- Items followed by an ellipsis (. . .) may be repeated any number of times (up to the length of the line).

AUTO

Format: AUTO [*line number*[,*increment*]]

Purpose: To generate a line number automatically after every carriage return.

Remarks: AUTO begins numbering at *line number* and increments each subsequent line number by *increment*. The default for both values is 10. If *line number* is followed by a comma but *increment* is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing <CTRL>C. The line in which <CTRL>C is typed is not saved. After <CTRL>C is typed, BASIC returns to command level.

Example:

```
AUTO 100,50      Generates line numbers 100, 150, 200 . . .
AUTO             Generates line numbers 10, 20, 30, 40 . . .
```

Note: This statement is named SEQUENCE or SEQ in DEC BASIC.

CLEAR

Format: CLEAR [,,*expression*]

Purpose: To set all numeric variables to zero and all string variables to null; and, optionally, to set the amount of stack space.

Remarks: *expression* sets aside stack space for BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

Examples:

```
CLEAR
CLEAR ,,2000
```

Note: This statement has an effect similar to the DEC BASIC SCRATCH statement.

CONT

Format: CONT

Purpose: To continue program execution after a <CTRL>C has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number.

CONT is invalid if the program has been edited during the break.

Example: See example of STOP command.

<CTRL>C <CTRL>O <RCTRL>C <RCTRL>O

Format: [R]CTRL*char*
where *char* is either C or O

Purpose: <CTRL>C and <CTRL>O disable the checking for <CTRL>C and <CTRL>O, respectively, for the running program. <RCTRL>C and <RCTRL>O re-enable the checking for <CTRL>C and <CTRL>O, respectively.

When checking for a character is disabled, that character may be read as data. When GIGI BASIC is first accessed, or whenever a NEW command is given, checking for both <CTRL>C and <CTRL>O is enabled.

Checking for these control characters can be re-established individually by the use of the <RCTRL>C and <RCTRL>O commands.

Remarks: If both <CTRL>C and <CTRL>O are in effect, it is possible to type-ahead well over 200 characters of input to the program; but if the checking for either is in effect, only the most recent typed-ahead character is retained for input to the program.

When <CTRL>C is in effect, the running program cannot be stopped by the <CTRL>C combination (unless the program itself is checking for it). In every case, however, the SHIFT-RESET combination will stop the GIGI BASIC program.

Note: These are functions in DEC BASIC.

DATA

Format: DATA *list of constants*

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ, below.)

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

list of constants may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement (see RESTORE, below).

Example: See examples of READ command.

DEF FN

Format: DEF FN*name*[(*parameter*)] = *function definition*

Purpose: To define and name a function that is written by the user.

Remarks: *name* must be a legal variable name. This name, preceded by FN, becomes the name of the function. *parameter* is the variable name in the function definition that is to be replaced when the function is called. *function definition* is an expression that performs the operation of the function. It is limited to one line. The parameter name that appears in this expression serves only to define the function; it does not affect a program variable having the same name. A variable name used in a function definition may or may not appear as the parameter. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

User-defined string functions are not allowed.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an UNDEFINED USER FUNCTION error occurs. DEF FN is illegal in the direct mode.

Example:

```
.
.
410     DEF FNAB(X) = X^3/A^2
420     T = FNAB(I)
.
.
```

Line 410 defines the function FNAB. The function is called in line 420.

DELETE

Format: DELETE[*line number*][– *line number*]

Purpose: To delete program lines.

Remarks: GIGI BASIC always returns to command level after a DELETE is executed. If *line number* does not exist, an ILLEGAL FUNCTION CALL error occurs.

Examples:

DELETE 40	Deletes line 40
DELETE 40 – 100	Deletes lines 40 through 100, inclusive
DELETE – 40	Deletes all lines up to and including line 40

DIM

Format: DIM *list of subscripted variables*

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a SUBSCRIPT OUT OF RANGE error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see below.)

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:

```
10      DIM A(20)
20      FOR I=0 TO 20
30      READ A(I)
40      NEXT I
          .
          .
          .
```

ECHO/NOECHO

Format: [NO]ECHO [#*n*]where *n* is a channel number (see INPUT statement)

Purpose: NOECHO disables echoing of input to the implied or expressed data channel.

ECHO enables echoing of input to the specified or implied channel.

Remarks: If the channel number is not specified, it defaults to 0 (zero) — which is the interpreter input channel and likewise the default for the INPUT and LINPUT commands.

Normally, input from channels not connected to the host is immediately echoed, or sent back, as output over that same channel. This allows a typist supplying input to be able to see what is being typed. When echo is disabled for a channel, the characters sent to that channel are not immediately displayed to that channel, and a typist would get no feedback in that case.

Echoing can be harmful when done to a non-human sender, such as the host, if that sender is not expecting it (which is usually the case). That is why input received from the host is never echoed back to the host by GIGI (there is no way to turn on such echo). For the same reason, when transmitting to the host from a GIGI BASIC program, it is often desirable to turn off echoing by the host. Such a function depends on the host being used, however, and cannot be covered in this manual.

It is sometimes necessary to turn off echo to the terminal if it is desired to read the keyboard or a ReGIS report while the terminal is in graphics mode (see the GON$ and GOFF$ functions). The echoing of input in such cases may be misinterpreted as commands by the ReGIS processor, and they wouldn't be readable by the terminal user, anyway.

For information on channel numbers, see also the INPUT statement.

EDIT

Format: EDIT *line number*

Purpose: To enter Edit Mode at the specified line.

Remarks: In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, GIGI BASIC types the line number of the line to be edited, then it types a space and waits for an Edit Mode subcommand.

Edit Mode Subcommands

Edit Mode subcommands are used to move the cursor or to insert, delete, or replace text within a line. The subcommands are not echoed.

Edit Mode subcommands may be categorized according to the following functions:

- Moving the cursor
- Inserting text
- Deleting text
- Ending and restarting Edit Mode

Note: In the descriptions that follow, *ch* represents any character, *text* represents a string of characters of arbitrary length, and $ represents the Escape key.

Moving the Cursor

Space: Use the space bar to move the cursor to the right. Characters are printed as you space over them.

BackSpace in Edit Mode: BackSpace moves the cursor i spaces to the left. Characters are printed as you backspace over them.

Inserting Text

I: I *text*$ inserts *text* at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, type Escape. If Carriage Return is typed during an Insert command, the effect is the same as typing Escape and then Carriage Return. During an Insert command, the Delete key on the terminal may be used to delete characters to the left of the cursor. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (<CTRL>G) is typed and the character is not printed.

Deleting Text

Delete: The Delete key deletes i characters to the left of the cursor. The deleted characters disappear from the screen, and the cursor is positioned to the left of the last character deleted.

Ending and Restarting Edit Mode

<CR>: Typing Carriage Return prints the remainder of the line, saves the changes you made and exits Edit Mode.

<CTRL>C: The <CTRL>C subcommand returns to GIGI BASIC command level, without saving any of the changes that were made to the line during Edit Mode.

L: The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in Edit Mode. L is usually used to list the line when you first enter Edit Mode.

Note: If GIGI BASIC receives an unrecognizable command or illegal character while in Edit Mode, it prints a bell (<CTRL>G) and the command or character is ignored.

END

Format: END

Purpose: To terminate program execution and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. GIGI BASIC always returns to command level after an END is executed.

Example:

```
520 IF K > 1000 THEN END ELSE GOTO 20
```

Note: An END may occur only as the last statement on the highest numbered line in a program under DEC BASIC.

ERASE

Format: ERASE *list of array variables*

Purpose: To eliminate arrays from a program.

Remarks: Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purpose. If an attempt is made to redimension an array without first ERASEing it, a REDIMENSIONED ARRAY error occurs.

Example:

```
.
.
.
450     ERASE  A,B
460     DIM B(99)
.
.
.
```

ERR AND ERL VARIABLES

When an error handling subroutine is entered, the variable ERR contains the error code for the variable and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF. . . THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use

IF 65535 = ERL THEN . . .

Otherwise, use

IF ERR = *error code* THEN . . .
IF ERL = *line number* THEN . . .

Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. GIGI BASIC's error codes are listed in Appendix A.

ERROR

Format: ERROR *integer expression*

Purpose: 1) To simulate the occurrence of a GIGI BASIC error; or 2) to allow error codes to be defined by the user.

Remarks: The value of *integer expression* must be greater than 0 and less than 255. If the value of *integer expression* equals an error code already in use by GIGI BASIC (see Appendix A), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by GIGI BASIC's error codes. (It is preferable to use the highest available values.) This user-defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, GIGI BASIC responds with the message ?UE ERROR. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1:

```
LIST
10      S = 10
20      T = 5
30      ERROR S + T
40      END
Ok
RUN
?LS Error in line 30
```

Or, in direct mode:

```
Ok ERROR 15              (you type this line)
?LS Error                (GIGI BASIC types this line)
Ok
```

Example 2:

```
  .
  .
  .
110     ON  ERROR  GOTO  400
120     INPUT  WHAT  IS  YOUR  BET;B
130     IF  B  >  5000  THEN  ERROR  210
  .
  .
  .
400     IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410     IF ERL = 130 THEN RESUME 120    .
  .
  .
```

FOR . . . NEXT

Format:

FOR *variable* = x TO y [STEP z]
.
.
.
NEXT [*variable*][,*variable*. . .]

where x, y and z are numeric expressions.

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: *variable* is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, GIGI BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR . . . NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Nested Loops

FOR . . . NEXT loops may be nested, that is, a FOR . . . NEXT loop may be placed within the context of another FOR . . . NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop.

If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a ?NF error message is issued and execution is terminated.

Example 1:

```
10      K = 10
20      FOR I = 1 TO K STEP 2
30      PRINT I;
40      K = K + 10
50      PRINT K
60      NEXT
RUN
  1     20
  3     30
  5     40
  7     50
  9     60
Ok
```

Example 2:

```
10      J=0
20      FOR I=1 TO J
30      PRINT I
40      NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3:

```
10      I=5
20      FOR I=1 TO I+5
30      PRINT I;
40      NEXT
RUN
  1     2     3     4     5     6     7     8     9     10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set.

GOSUB . . . RETURN

Format:

GOSUB *line number*

.
.
.

RETURN

Purpose: To branch to and return from a subroutine.

Remarks: *line number* is the first line of the subroutine. A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause GIGI BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

Example:

```
10      GOSUB 40
20      PRINT " BACK FROM SUBROUTINE"
30      END
40      PRINT " SUBROUTINE";
50      PRINT " IN";
60      PRINT " PROGRESS"
70      RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

GOTO

Format: GOTO *line number*

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If *line number* is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after *line number.*

Example:

```
LIST
10      READ R
20      PRINT R =;R,
30      A = 3.14*R^2
40      PRINT AREA =;A
50      GOTO 10
60      DATA 5,7,12
Ok
RUN
R  =  5                 AREA  =  78.5
R  =                    AREA  =  153.86
R  =  12                AREA  =  452.16
?OD Error in 10
Ok
```

HOST

Format: HOST

Purpose: To terminate interaction with GIGI BASIC and restore interaction with the host computer.

Remarks: This command allows the user to return to communicating directly with the host computer without having to change the BASIC set-up mode to BA0. It also transmits the line REM HOST to the host computer.

IF . . . THEN[. . . ELSE] and IF . . . GOTO

Format 1: IF *expression* THEN *statement(s)* | *line number*

[ELSE *statement(s)* | *line number*]

Format 2: IF *expression* GOTO *line number*

[ELSE *statement(s)* | *line number*]

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of *expression* is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of *expression* is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement.

Nesting of IF Statements

IF . . . THEN . . . ELSE statements may be nested. Nesting is limited only by the length of the line. For example:

```
IF X>Y THEN PRINT "GREATER ELSE" IF Y>X
        THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example:

```
IF A=B THEN IF B=C THEN PRINT "A=C"
        ELSE PRINT "A<>C"
```

will not print A<>C when A<>B.

If an IF . . . THEN statement is followed by a line number in the direct mode, an "?UL" error results unless a statement with the specified line number had previously been entered in the indirect mode.

Note: When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN . . .
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 2:

```
100     IF(I<20)*(I>10) THEN DB=1979-1:GOTO 300
110     PRINT "OUT OF RANGE"
          .
          .
          .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3:

```
210     IF IOFLAG THEN PRINT A$ ELSE PRINT #3, A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

INPUT

Format: INPUT [#N,][;][*"prompt string"*;] *list of variables*
where N is a channel number

Purpose: To allow input from the terminal or specified channel during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If *"prompt string"* is included, the string is printed before the question mark. The required data is then entered at the terminal or specified channel.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/line feed sequence.

If echoing is turned off on the implied or specified channel by means of the NOECHO statement; the prompt, if any, is issued, but none of the input characters is echoed.

The data that is entered is assigned to the variable(s) given in *variable list*. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUTwith too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

INPUT is illegal in the direct mode.

I/O Channels

INPUT normally receives data from the same I/O channel used by the interpreter for receiving command and program lines. This is called channel 0 (zero), and is connected to the keyboard when in "local BASIC mode" (SET-UP BA1) or to the host communication line when in "host BASIC mode" (SET-UP BA2). The channel number to use can be stated by "#N", where N is a channel number from 0 (zero) to 3, after the word INPUT. Channel #1 is always the terminal; channel #2 is always the host communication line; and channel #3 is always the auxiliary (hardcopy/tablet) port.

Examples:

```
10      INPUT X
20      PRINT X " SQUARED IS"  X^2
30      END
RUN
?       5       (The 5 was typed in by the user
8               in response to the question mark.)
  5     SQUARED IS 25
Ok

LIST
10      PI = 3.14
20      INPUT " WHAT IS THE RADIUS" ;R
30      A = PI * R^2
40      PRINT " THE AREA OF THE CIRCLE IS " ;A
50      PRINT
60      GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4    (User types 7.4)
THE AREA OF THE CIRCLE is 171.946

WHAT IS THE RADIUS?
              .
              .
              .
```

LET

Format: [LET] *variable = expression*

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110     LET D = 12
120     LET E = 12^2
130     LET F = 12^4
140     LET SUM = D + E + F
              .
              .
              .
            or
110     D = 12
120     E = 12^2
130     F = 12^4
140     SUM = D + E + F
              .
              .
              .
```

LINPUT

Format: LINPUT[#N,][;][*PROMPT STRING*;]*string variable*
where N is a channel number (see INPUT statement).

Purpose: To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks: The prompt string is a string literal that is printed at the terminal before input is accepted.

A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to *string variable*.

If LINPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

If the NOECHO statement has been issued for the specified or implied channel, then none of the characters typed by the user is echoed.

A LINPUT may be escaped by typing <CTRL>C. GIGI BASIC will return to command level and type Ok. Typing CONT resumes execution at the LINPUT.

Example:

```
20      LINPUT CUSTOMER INFORMATION? ;C$
30      PRINT C$
60      LINPUT #2, C$
70      PRINT C$
RUN
CUSTOMER INFORMATION?  LINDA JONES      234,4      MEMPHIS
LINDA JONES      234,4      MEMPHIS
Ok
```

LIST

Format: LIST [#n,] [*line number*[– [*line number*]]]
where *n* is a channel number (see INPUT statement).

Purpose: To list all or part of the program currently in memory at the terminal.

Remarks: GIGI BASIC always returns to command level after a LIST is executed.

If *line number* is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing <CTRL>C.) If only *line number* is included, only the specified line is listed.

- If only the first number is specified, that line and all higher-numbered lines are listed.
- If only the second number is specified, all lines from the beginning of the program through that line are listed.
- If both numbers are specified, the entire range is listed.

Examples:

LIST	Lists the program currently in memory.
LIST 500	Lists line 500.
LIST 150 –	Lists all lines from 150 to the end.
LIST – 1000	Lists all lines from the lowest number through 1000.
LIST 150 – 1000	Lists lines 150 through 1000, inclusive.
LIST #3	Lists the program currently in memory on the hardcopy unit.
LIST #2, 200 –	Sends program lines from 200 through the end of the program to the host (in general, if the host is echoing input, a deadlock situation can result).

MID$

Format: MID$(*string exp1*,n[,m]) = *string exp2*

where n and m are integer expressions and *string exp1* and *string exp2* are string expressions.

Purpose: To replace a portion of one string with another string.

Remarks: The characters in *string exp1*, beginning at position n, are replaced by the characters in *string exp2*. The optional m refers to the number of characters from *string exp2* that will be used in the replacement. If m is omitted, all of *string exp2* is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of *string exp1*.

Example:

```
10      A$ = " KANSAS CITY, MO"
20      MID$(A$,14) = " KS"
30      PRINT A$
RUN
KANSAS CITY, KS
```

MID$ may also be used as a function that returns a substring of a given string. See Chapter 3.

Note: DEC BASIC allows MID$ only on the right hand side of an assignment.

NEW

Format: NEW

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. GIGI BASIC always returns to command level after a NEW is executed.

OLD

Format: OLD *string constant or expression*

Purpose: To request a copy of a BASIC program from the host computer.

Remarks: The functioning of this command depends upon the program running at the host computer. The *string constant or expression* specifies the file name of the GIGI BASIC program.

See also the companion SAVE statement, for storing a GIGI BASIC program on the host.

Note: The following describes the terminal-host interaction initiated by the OLD command.

Upon receipt of the OLD command, a NEW command is simulated so that the current program and variables are deleted, and then GIGI BASIC sends the line:

REM OLD *character data*

to the host, where *character data* is the evaluation of the argument of the OLD command (without quotes). GIGI BASIC then enters a mode similar to normal direct mode, but where it is reading input from the host rather than from the terminal. It is then expected that the host will transmit program lines (lines starting with line numbers) to GIGI. Upon receipt of the first line not starting with a line number, it is executed as a direct mode command. When the execution of that direct mode line is completed, GIGI BASIC returns normal command to the terminal.

The OLD command is intended to be used only in LOCAL BASIC mode (BA1). The host computer must not echo lines sent to it.

ON ERROR GOTO

Format: ON ERROR GOTO *line number*

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Remarks: Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine.

If *line number* does not exist, an UNDEFINED LINE error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes GIGI BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Note: If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

Example:

```
10 ON ERROR GOTO 1000
```

ON . . . GOSUB and ON . . . GOTO

Format: ON *expression* GOTO *list of line numbers*

ON *expression* GOSUB *list of line numbers*

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of *expression* determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON . . . GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of *expression* is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of *expression* is negative or greater than 255, an ?FC error occurs.

Example:

```
100 ON L-1 GOTO 150,300,320,390
```

OPTION BASE

Format: OPTION BASE n
where n is 1 or 0

Purpose: To declare the minimum value for array subscripts.

Remarks: The default base is 0. If the statement

OPTION BASE 1

is executed, the lowest value an array subscript may have is one.

OUT

Format: OUT I,J
where I and J are integer expressions in the range 0 to 255.

Purpose: To send a byte to a machine output port.

Remarks: The integer expression I is the port number, and the integer expression J is the data to be transmitted.

Example:

```
100 OUT 32,100
```

PRINT

Format: PRINT [#N,] [*list of expressions*]
where N is a channel number (see INPUT statement.)

Purpose: To output data at the terminal or specified channel.

Remarks: If *list of expressions* is omitted, a blank line is printed. If *list of expressions* is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. GIGI BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value.

Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, GIGI BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 10^(-6) is output as .000001 and 10^(-7) is output as 1E-7. Also, 10^(-16) is output as .0000000000000001 and 10^(-17) is output as 1E-17.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1:

```
10      X=5
20      PRINT X+5, X-5, X*(-5), X^5
30      END
RUN
10              0               -25             3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2:

```
LIST
10      INPUT X
20      PRINT X SQUARED IS X^2 AND;
30      PRINT X CUBED IS X^3
40      PRINT
50      GOTO 10
Ok
RUN
? 9
   9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
   21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3:

```
10      FOR X = 1 TO 5
20      J=J+5
30      K=K+10
40      ?J;K;
50      NEXT X
Ok
RUN
  5     10      10      20      15      30      20      40      25      50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

RANDOMIZE

Format: RANDOMIZE [*expression*]

Purpose: To reseed the random number generator.

Remarks: If *expression* is omitted, GIGI BASIC uses an internally-generated 16-bit value, incremented every 1/60th of a second, as the new random number generator seed.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example:

```
10      RANDOMIZE
20      FOR I = 1 TO 5
30      PRINT RND;
40      NEXT I
RUN
.88598 .484668 .586328 .119426 .709225
Ok
RUN
.803506 .162462 .929364 .292443 .322921
Ok
```

READ

Format: READ *list of variables*

Purpose: To read values from a DATA statement and assign them to variables. (See DATA, above.)

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a SYNTAX ERROR will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in *list of variables* exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see RESTORE, below.)

Example 1:

```
.
.
.
80      FOR I = 1 TO 10
90      READ A(I)
100     NEXT I
110     DATA 3.08,5.19,3.12,3.98,4.24
120     DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2:

```
LIST
10      PRINT " CITY", " STATE" , " ZIP"
20      READ C$,S$,Z
30      DATA LITTLETON,, COLORADO, 80123
40      PRINT C$,S$,Z
Ok
RUN
CITY            STATE           ZIP
LITTLETON,      COLORADO        80123
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

REM

Format: REM *remark*

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

Example:

```
.
.
.
120    REM  CALCULATE  AVERAGE  VELOCITY
130    FOR I=1 TO 20
140    SUM=SUM + V(I)
.
.
.

.
.
.
120    FOR  I=1  TO  20     'CALCULATE  AVERAGE  VELOCITY
130    SUM=SUM+V(I)
140    NEXT I
.
.
.
```

Note: DEC BASIC uses the exclamation point character in the same manner GIGI BASIC uses the single quote (apostrophe).

RESTORE

Format: RESTORE [*line number*]

Purpose: To allow DATA statements to be reread from a specified point.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program.

If *line number* is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:

```
10    READ A,B,C
20    RESTORE
30    READ D,E,F
40    DATA 57, 68, 79
.
.
.
```

RESUME

Format:

RESUME

RESUME 0

RESUME NEXT

RESUME *line number*

Purpose: To continue program execution after an error recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME *line number*	Execution resumes at *line number*.

A RESUME statement that is not in an error trap routine causes a ?RW ERROR message to be printed.

Example:

```
10      ON ERROR GOTO 900
   .
   .
   .
900
IF (ERR = 230)AND(ERL = 90) THEN PRINT " TRY AGAIN" :RESUME 80
   .
   .
   .
```

RUN

Format: RUN [*line number*]

Purpose: To execute the program currently in memory.

Remarks: If *line number* is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. GIGI BASIC always returns to command level after a RUN is executed.

Example:

```
RUN
```

SAVE

Format: SAVE *string constant or expression*

Purpose: To send a copy of the GIGI BASIC program to the host computer.

Remarks: The functioning of this command depends upon the program running at the host computer. The *string constant or expression* specifies the file name of the GIGI BASIC program.

See also the companion OLD statement.

Note: The following describes the terminal-host interaction initiated by the SAVE command.

Upon receipt of the SAVE command, GIGI BASIC sends the line:

REM SAVE *character data*

to the host, where *character data* is the evaluation of the argument of the SAVE command. GIGI BASIC then transmits the program in memory to the host, as if a LIST 2 command were issued. The program list is then followed by the following line:

REM END SAVE

also sent to the host. GIGI BASIC then enters a mode similar to normal direct mode, but where it is reading input from the host rather than from the terminal. Upon receipt of the first line not starting with a line number, it is executed as a direct mode command (warning: lines starting with line numbers will be stored in memory). That command can be a print statement which reports on the success or failure of the SAVE operation. When the execution of that direct mode line is completed, GIGI BASIC returns normal command to the terminal.

The SAVE command is intended to be used only in LOCAL BASIC mode (BA1). The host computer must not echo lines sent to it.

STOP

Format: STOP

Purpose: To terminate program execution and return to command level.

Remarks: STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

Break in line nnnnn

Unlike the END statement, the STOP statement does not close files.

GIGI BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see above.)

Example:

```
10      INPUT A,B,C
20      K=A^2*5.3:L=B^3/.26
30      STOP
40      M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
30.7692
Ok
CONT
115.9
Ok
```

SWAP

Format: SWAP *variable,variable*

Purpose: To exchange the values of two variables.

Remarks: Any type variable may be SWAPped (numeric or string), but the two variables must be of the same type or a ?TM error results.

Example:

```
LIST
10      A$=" ONE " : B$=" ALL " : C$="FOR"
20      PRINT A$ C$ B$
30      SWAP A$, B$
40      PRINT A$ C$ B$
RUN
Ok
ONE FOR ALL
ALL FOR ONE
Ok
```

TRON/TROFF

Format:

TRON

TROFF

Purpose: To trace the execution of program statements.

Remarks: As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed.

The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example:

```
TRON
Ok
LIST
10      K=10
20      FOR J=1 TO 2
30      L=K + 10
40      PRINT J;K;L
50      K=K+10
60      NEXT
70      END
Ok
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
```

WAIT

Format:

WAIT *port number*, I[,J]
where I and J are integer expressions.

Purpose: To suspend program execution while monitoring the status of a machine input port.

Remarks: The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, GIGI BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

CAUTION: It is possible to enter an infinite loop with the WAIT statement, in which case a hard RESET will be necessary.

Example:

```
100 WAIT 32,2
```

Note: The WAIT statement in DEC BASIC is totally different from the GIGI BASIC WAIT.

WHILE . . . WEND

Format:

WHILE *expression*

.
.

[*loop statements*]

.
.

WEND

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Remarks: If *expression* is not zero (i.e., true), *loop statements* are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks *expression*. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a ?WH error, and an unmatched WEND statement causes a ?WE error.

Example:

```
90      BUBBLE SORT ARRAY A$
100     FLIPS = 1 'FORCE ONE PASS THRU LOOP
110     WHILE FLIPS
115                 FLIPS = 0
120                 FOR  I = 1  TO  J - 1
130                         IF  A$(I) > A$(I + 1)  THEN
                                    SWAP  A$(I),A$(I + 1):FLIPS = 1
140                 NEXT  I
150 WEND
```

Note: DEC BASIC uses NEXT instead of WEND.

WIDTH

Format:

WIDTH *integer expression*

Purpose: To set the printed line width in number of characters for the terminal or printer.

integer expression must have a value in the range 15 to 255. The default width is 255 characters.

If *integer expression* is 255, the line width is infinite, that is, BASIC never inserts a carriage return. However, the position of the cursor or the print head, as give n by the POS function, returns to zero after position 255.

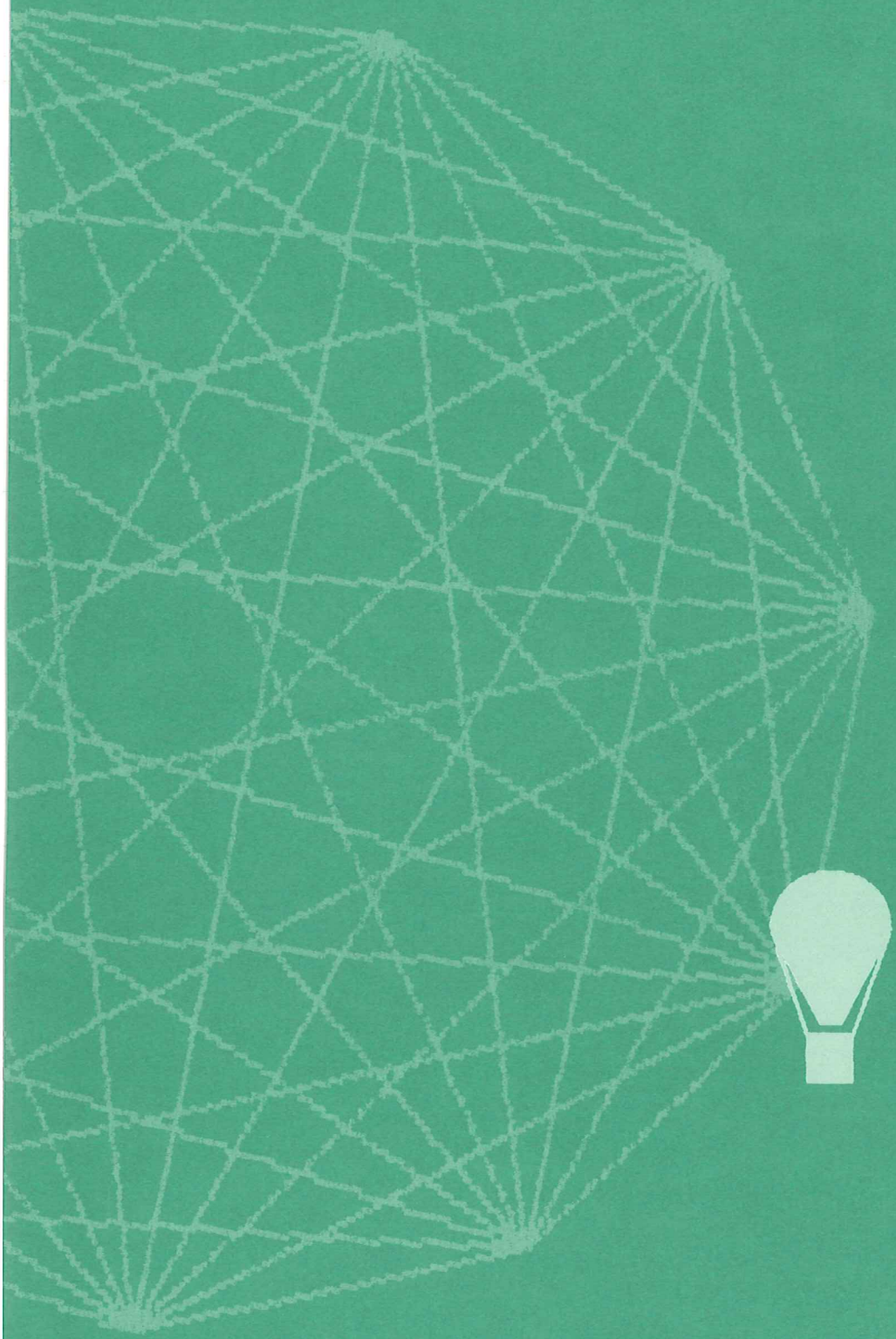
Example:

```
10      PRINT " ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNOPQR
STUVWXYZ
Ok
```

Note: See MARGIN in DEC BASIC.

3

GIGI BASIC Functions



3 GIGI BASIC FUNCTIONS

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

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

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

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

ABS

Format: ABS(X)

Action: Returns the absolute value of the expression X.

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

ASC

Format: ASC(X$)

Action: Returns a numerical value that is the ASCII code of the first character of the string X$. (See Appendix C for ASCII codes.) If X$ is null, a ?FC error is returned.

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

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

Note: This function is named ASCII in DEC BASIC. A null value for X$ results in 0 in DEC BASIC.

ATN

Format: ATN(X)

Action: Returns the arctangent of X in radians. Result is in the range –pi/2 to pi/2. The expression X must be numeric; the evaluation of ATN is performed with 24 bits of precision.

Example:

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

CHR$

Format: CHR$(I)

Action: Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix C.) CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message, or a form feed could be sent (CHR$(12)) to clear a CRT screen and return the cursor to the home position.

Example:

```
PRINT CHR$(66)
B
Ok
```

See the ASC function for ASCII-to-numeric conversion.

COS

Format: COS(X)

Action: Returns the cosine of X in radians. The evaluation of COS is performed with 24 bits of precision.

Example:

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

ESC$

Format: ESC$

Action: Returns a string of length one containing the ESCAPE character. This is the same character as is generated by CHR$(27).

EXP

Format: EXP(X)

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

Example:

```
10      X = 5
20      PRINT EXP (X – 1)
RUN
54.5982
Ok
```

FRE

Format: FRE(0)
FRE(X$)

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

FRE("") forces a garbage collection before returning the number of free bytes. BE PATIENT: garbage collection may take 1 to 1-1/2 minutes. BASIC will not initiate garbage collection until all free memory has been used up. Therefore, using FRE("") periodically will result in shorter delays for each garbage collection.

Example:

```
PRINT FRE(0)
14542
Ok
```

GOFF$

Format: GOFF$

Action: Generates a string of length 2 that contains the graphics off sequence, an escape sequence that causes GIGI to leave graphics mode. This sequence must be sent to GIGI with a PRINT statement to take effect.

The graphics off sequence consists of the characters ESCAPE and \ (backslash) in that order. This is also called the ANSI STRING TERMINATOR (ST) sequence; it can also be used to terminate the special strings used to change set-up settings and program the programmable keypad.

To enter graphics mode, use the GON$ string in a PRINT statement to send the special GRAPHICS ON control string to GIGI.

GON$

Format: GON$

Action: Returns the special GRAPHICS ON control string which, if sent to GIGI with a PRINT statement, will put GIGI into graphics mode (i.e., direct the ReGIS interpreter to parse any following data as ReGIS command string).

This function generates the 3-character string consisting of the ESCAPE code followed by Pp. When GIGI receives this code, it enters graphics mode and remains in graphics mode until the GRAPHICS OFF sequence is received.

This form of graphics mode is known as DCS (DEVICE CONTROL string). There is another graphics mode, called GRAPHICS PREFIX MODE, available to the GIGI user. This mode does not use the GON$ or GOFF$ codes.

HEX$

Format: HEX$(X)

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

Example:

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

See the OCT$ function for octal conversion.

INP

Format: INP(I)

Action: Returns the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement, Chapter 2.

Example:

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

INKEY$

Format: INKEY$[#N][W]

Action: Returns a string of 1 character, read from the terminal or from channel number N. No characters will be echoed and all control characters are passed through except <CTRL>C, if <CTRL>C checking is enabled (See the <CTRL>C statement) If the W is specified, the program halts until a character is available; if the W is not specified, INKEY$ always returns immediately to the program, either with the character, if available, or a null string.

Example:

```
.
.
.
100     PRINT "TYPE P TO PROCEED OR S TO STOP"
110     X$ = INKEY$W
120     IF X$ = P THEN 500
130     IF X$ = S THEN 700 ELSE 100
.
.
.
```

Note: This function is named ONECHR in DEC BASIC.

INSTR

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

Action: Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in the range 0 to 255.

If I > LEN(X$) or if X$ is null or if Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions or string literals.

Example:

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

INT

Format: INT(X)

Action: Returns the largest integer < = X.

Examples:

```
PRINT INT(99.89)
99
Ok
PRINT INT(-12.11)
-13
Ok
```

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

LEFT$

Format: LEFT$(X$,I)

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

Example:
```
10      A$ = "GIGI BASIC"
20      B$ = LEFT$(A$,4)
30      PRINT B$
GIGI
Ok
```

Also see the MID$ and RIGHT$ functions.

LEN

Format: LEN(X$)

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

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

LOG

Format: LOG(X)

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

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

MID$

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

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

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

Also see the LEFT$ and RIGHT$ functions.

OCT$

Format: OCT$(X)

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

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

See the HEX$ function for hexadecimal conversion.

POS

Format: POS(I)

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

Example:
```
IF POS(X) > 60 THEN PRINT CHR$(13)
```

Also see the LPOS function.

Note: This function is named CCPOS in DEC BASIC.

RIGHT$

Format: RIGHT$(X$,I)

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

Example:
```
10      A$ = " This is GIGI BASIC"
20      PRINT RIGHT$(A$,10)
RUN
GIGI BASIC
Ok
```

Also see the MID$ and LEFT$ functions.

RND

Format: RND[(X)]

Action: Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see RANDOMIZE, Chapter 2).

However, X < 0 always restarts the same sequence for any given X.

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

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

SGN

Format: SGN(X)

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

Example: ON SGN(X)+2 GOTO 100,200,300

branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.

SIN

Format: SIN(X)

Action: Returns the sine of X in radians. The evaluation of SIN is performed with 24 bits of precision. COS(X)=SIN(X+3.14159/2).

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

SPACE$

Format: SPACE$(X)

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

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

Also see the SPC function.

SPC

Format: SPC(I)

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

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

Also see the SPACE$ function.

Note: Refer to TAB and SPACE functions in DEC BASIC.

SQR

Format: SQR(X)

Action: Returns the square root of X. X must be >= 0.

Example:

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

STR$

Format: STR$(X)

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

Example:

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

Also see the VAL function.

STRING$

Formats: STRING$(I,J)

STRING$(I,X$)

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

Example:

```
10      X$ = STRING$(10,45)
20      PRINT X$ MONTHLY REPORT X$
RUN
----------MONTHLY REPORT----------
Ok
```

TAB

Format: TAB(I)

Action: Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to that position on the next line.

Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255.

TAB may only be used in PRINT statements.

Example:

```
10      PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20      READ A$,B$
30      PRINT A$ TAB(25) B$
40      DATA "G. T. JONES","$25.00"
RUN
NAME                            AMOUNT

G.  T.  JONES                   $25.00
Ok
```

TAN

Format: TAN(X)

Action: Returns the tangent of X in radians. The evaluation of TAN is performed with 24 bits of precision. If TAN overflows, the ?OV error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

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

VAL

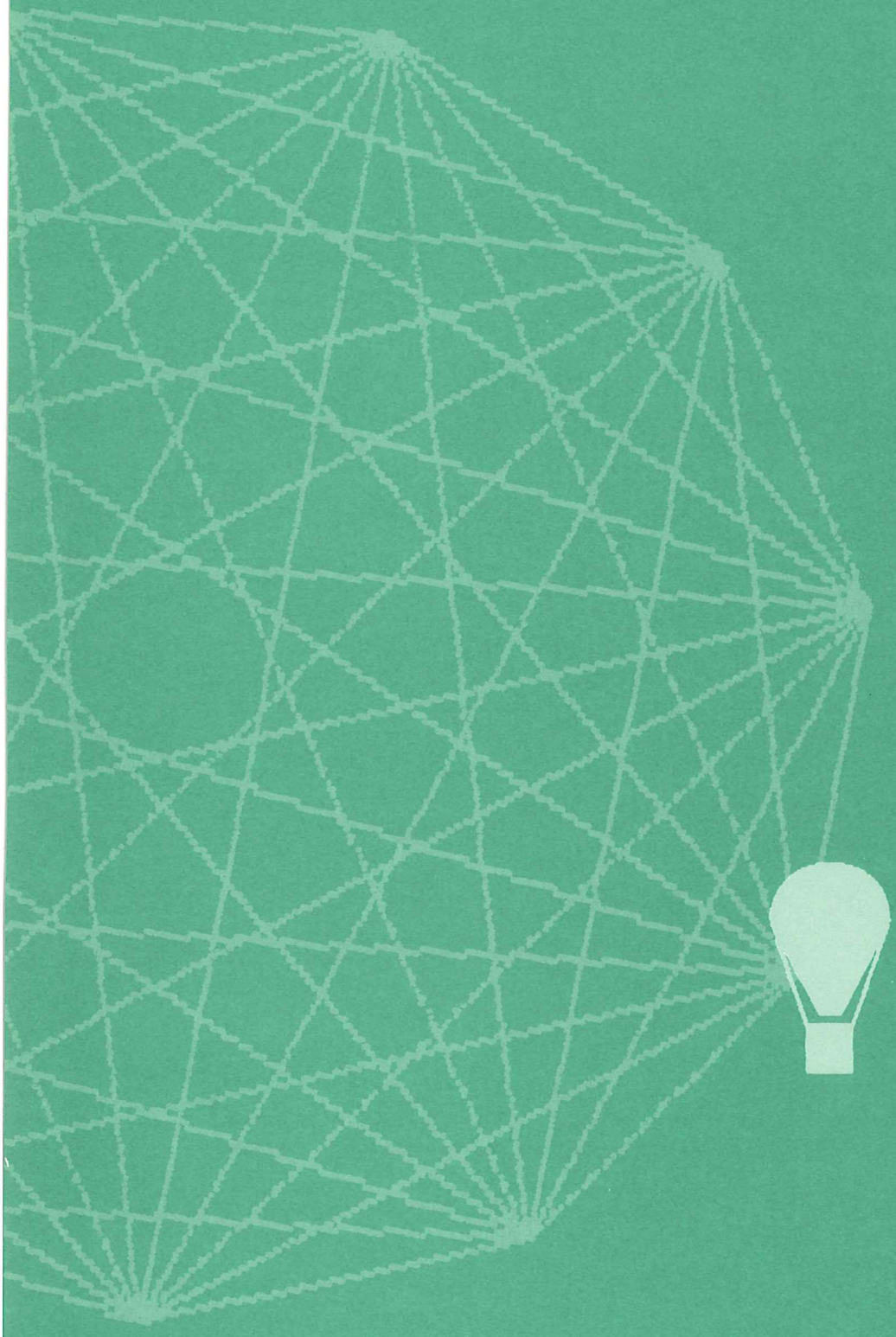
Format: VAL(X$)

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

Example:

```
10      READ NAME$,CITY$,STATE$,ZIP$
20      IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN PRINT
NAME$ TAB(25) "OUT OF STATE"
30      IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN PRINT
NAME$ TAB(25) "LONG BEACH"
        .
        .
        .
```

See the STR$ function for numeric-to-string conversion.



Appendices

APPENDIX A
Summary of Error Codes

Code	Number	Meaning
NF	**1**	**NEXT without FOR** A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
SN	**2**	**Syntax error** A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).
RG	**3**	**Return without GOSUB** A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
OD	**4**	**Out of data** A READ statement is executed when there are no DATA statements with unread data remaining in the program.
FC	**5**	**Illegal function call** A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: • a negative or unreasonably large subscript • a negative or zero argument with LOG • a negative argument to SQR • a negative mantissa with a non-integer exponent • a call to a USR function for which the starting address has not yet been given • an improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, TAB, SPC, STRING$, SPACE$, INSTR, or ON . . . GOTO . . .
OV	**6**	**Overflow** The result of a calculation is too large to be represented in GIGI BASIC's number format. If underflow occurs, the result is zero and execution continues without an error.
OM	**7**	**Out of memory** A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

Code	Number	Meaning
UL	8	**Undefined line** A line reference in a GOTO, GOSUB, IF . . . THEN . . . ELSE or DELETE is to a nonexistent line.
BS	9	**Subscript out of range** An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.
DD	10	**Redimensioned array** Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.
/0	11	**Division by zero** A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.
ID	12	**Illegal direct** A statement that is illegal in direct mode is entered as a direct mode command.
TM	13	**Type mismatch** A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.
OS	14	**Out of string space** String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.
LS	15	**String too long** An attempt is made to create a string more than 255 characters long.
ST	16	**String formula too complex** A string expression is too long or too complex. The expression should be broken into smaller expressions.
CN	17	**Can't continue** An attempt is made to continue a program that: • has halted due to an error, • has been modified during a break in execution, or • does not exist.

Code	Number	Meaning
UF	18	**Undefined user function** A USR function is called before the function definition (DEF statement) is given.
NR	19	**No RESUME** An error trapping routine is entered but contains no RESUME statement.
RW	20	**RESUME without error** A RESUME statement is encountered before an error trapping routine is entered.
UE	21	**Unprintable error** An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.
MO	22	**Missing operand** An expression contains an operator with no operand following it.
FN	23	**FOR without NEXT** A FOR was encountered without a matching NEXT . . .
??	24	**Internal GIGI BASIC error.**
WH	25	**WHILE without WEND** A WHILE statement does not have a matching WEND.
WE	26	**WEND without WHILE** A WEND was encountered without a matching WHILE.

APPENDIX B
Mathematical Functions

Derived Functions

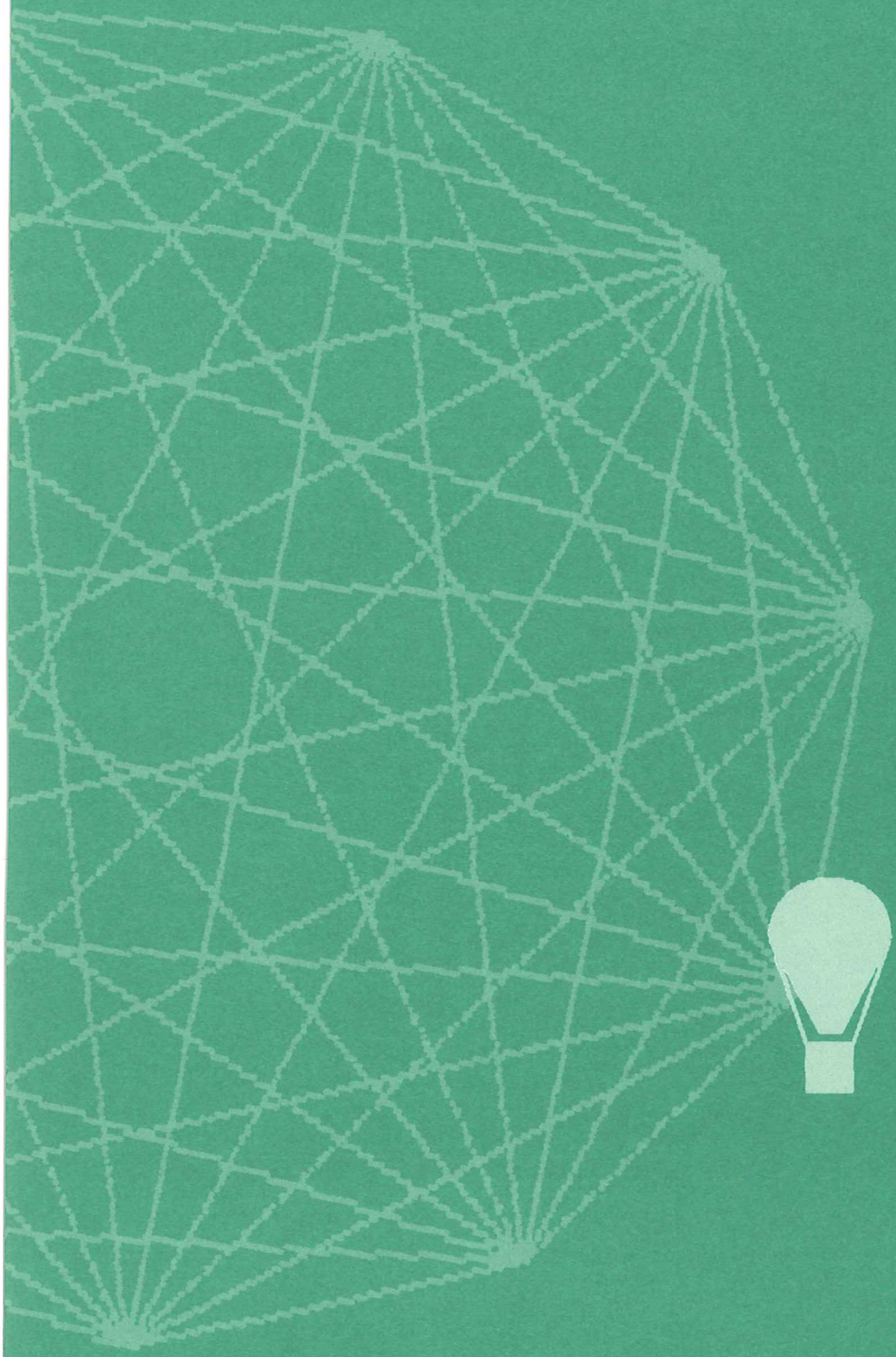
Functions that are not intrinsic to GIGI BASIC may be calculated as follows :

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

APPENDIX C
ASCII Character Codes

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

ASCII codes are in decimal. LF = Line Feed, FF = Form Feed, CR = Carriage Return, DEL = Delete BS = BackSpace



Index

INDEX

I

L

M

N

O

P

R

S

T

V

W

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

- Did you find errors in this manual? If so, specify by page.

- Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

- Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

- Please indicate the type of user/reader that you most nearly represent.

☐ Assembly language programmer
☐ Higher-level language programmer
☐ Occasional programmer (experienced)
☐ User with little programming experience
☐ Student programmer
☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

Please cut along this line.

Fold Here----------

Do Not Tear — Fold Here and Staple----------

No Postage
Necessary
If Mailed In The
United States

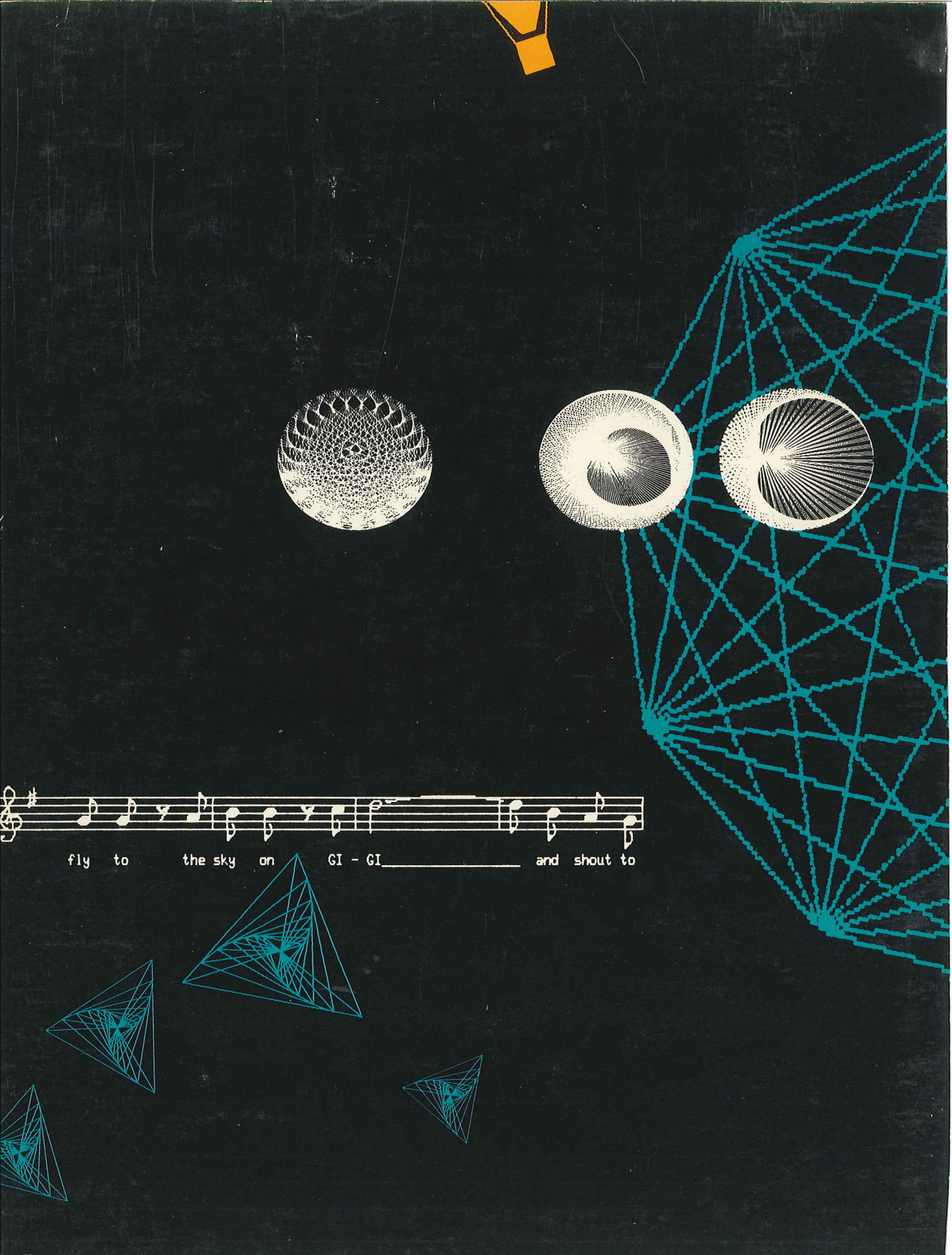
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 152 MARLBOROUGH, MA 01752

Postage will be paid by:

digital

Education Computer Systems
200 Forest Street MR1-1/E47
Marlborough, Massachusetts 01752


fly to the sky on GI - GI_____ and shout to