System 80

BASIC MANUAL

PREFACE

As the Computer Age gathers momentum, more and more people are becoming aware of the latest dramatic development: the microcomputer. To take full advantage of these incredibly powerful new tools, you need to be able to write programs. The easiest and most effective way to learn programming is by actually using the computer yourself, to get "hands-on" experience.

This manual is designed to help you learn computer programming on the System 80 Computer, using the "hands-on" approach. It provides a comprehensive software course, based on the Active Commands. Text Editing and program statements available in the System 80 powerful Level II BASIC language.

To get the most from the manual, we suggest very strongly that you read it from the first page to the last, without skipping any portion of the text or any of the numerous examples given. Not only that, but we suggest that you try each example for yourself on the System 80 computer as you go along. That way, you should really have a good grasp of programming by the time you reach the last page.

Happy and effective computing with your System 80!

TABLE OF CONTENTS

		page
	PREFACE	
	INTRODUCTION	2
1.	ACTIVE COMMANDS	12
2.	TEXT EDITING	20
3.	BASIC PROGRAMMING STATEMENTS	28
4.	PROCESSING ARRAYS	61
5.	STRING HANDLING	67
6.	BUILT-IN ARITHMETIC FUNCTIONS	73
7.	GRAPHICS FEATURES	76
8.	SPECIAL FEATURES	77
APPENDIX	A RESERVED WORDS	79
	B ERROR CODES	80
	C CONTROL, GRAPHICS, AND ASCII CODES	83
	D PROGRAM LIMITS	85
	E VIDEO DISPLAY MAP	86

INTRODUCTION

In the System 80, there are four operating levels:

(1) The Active Command Level: In this level, the computer responds to commands as soon as they are entered followed by hitting the NEW LINE key. Whenever the >_ signs are on the display, the user is in the Active Command level. (For more details see Chapter 1).

(2) The Program Execution level: This level is entered by typing RUN, and the BASIC program in the memory is executed. All numeric variables are set to zero and all string variables are set to null before execution starts, that is right after the RUN command is entered. (For more detail see the RUN command in Chapter 1).

(3) The Text Editing level: This level allows the user to modify, delete and add the content of the program in the memory. The user can change any part of the program text as desired, instead of retyping the entire program line.
(see Chapter 2).

(4) The Monitor level: This level permits the user to load machine language "object files" into memory. These object files (either program or data) can be accessed by other BASIC programs, or executed independently.
(see the SYSTEM command in Chapter 1).

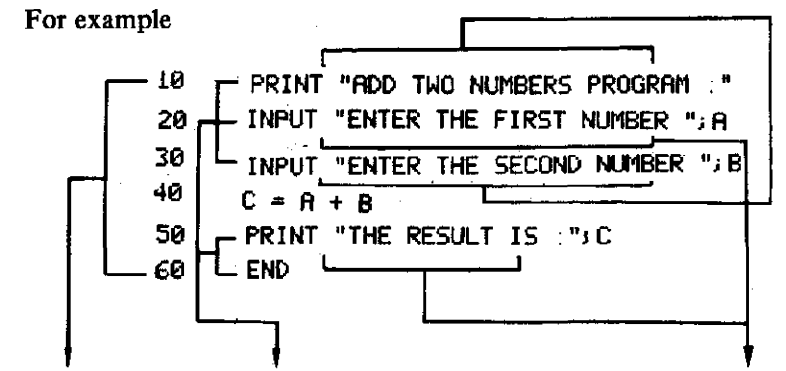
Before going into program coding, we should be familiar with some basic concepts of programming.

1. Keywords: There is a set of keywords (reserved words which form the skeleton of a program in Extended Basic. Some of the keywords are:

   PRINT
   INPUT
   IF
   THEN
   GOTO
   END

For the entire list of keywords, please refer to Appendix A. The keywords act as the guideline of a program.

For example



All this program does is to accept two numbers, add them together, and print out the result.

```
READY
>RUN

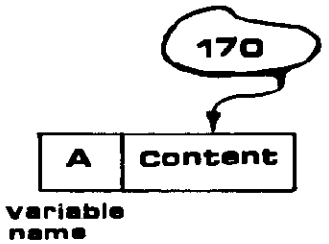
ADD TWO NUMBERS PROGRAM :
ENTER THE FIRST NUMBER ? 42
ENTER THE SECOND NUMBER ? 45
THE RESULT IS : 87
```

2. Variables: Do you remember your mail box number?
   Of course, right! The mail box number serves as a label which identifies itself from the others, so the postman can put the right letter to the right box.

The variable names (or identifiers) in a program function exactly like the mail box number. However, the variables use figurative names instead of numbers.

Let us consider the following events:

A = 170



(a)



(b)

In event (a), when the computer executes the assignment statement A = 170, it searches the location of variable A, and put the value of 170 into A's content. Just like in event (b), once the postman sees the address on the letter, he will bring the letter to the mail box 456, in ELM street. Simple, right!

With the same process, consider the following program.

```
10 A = 1          :REM   FILL THE CONTENT OF A WITH 1.
20 A = A + 10     :REM   ADD 10 TO A, STORE THE RESULT IN A.
30 B = A * 2      :REM   A X 2, STORE THE RESULT IN B.
40 PRINT "THE RESULTS ARE :"; A, B
50 END            :REM   END OF PROGRAM.
```

line 10

A	1

line 20

A	11

line 30

A	11
B	22

```
READY
>RUN
THE RESULTS ARE : 11            22
```

So far we only deal with variables that contain numbers; actually, variables may contain strings (one or more characters).

However, these variables are a little bit different from numeric variables.

```
10 A$ = "MR.  JOHN ADAMS,"
20 B$ = "P. O.  BOX 456,"
30 C$ = "ELM STREET."
40 PRINT A$
50 PRINT B$; C$
60 END


READY
>RUN

MR.  JOHN ADAMS,
P. O.  BOX 456, ELM STREET.
```

This program assigns "MR. JOHN ADAMS,", "P.O. BOX 456,", "ELM STREET." to A$, B$, C$ respectively. Then print out the contents of A$, B$ and C$ onto the screen.
Note that there is a $ sign following A, B and C. The "$" tells the computer that variables A$, B$, and C$ are string variables (variables that contain letters, symbols, as well as non – computational numbers). The value must be enclosed in double quotation marks. For example: D$ = "ABCDE 12345 * = /+"

Note that if you assign the wrong thing to the wrong variable, the computer will not understand and will give you an error message.

For example:

A = "WRONG DATA" (assign a string value to a numeric variable)

B$ = 100 (assign a numeric value to a string variable)

Besides, variable names must be unique. Just like two mail boxes cannot bear the same number.

The System 80 accepts variable names which are longer than two characters; however, only the first two characters are used by the computer to distinguish between other variables. Variable names must begin with a letter (from A to Z) and followed by another letter or a digit (from 0 to 9). The following are valid and distinct variables:

A, AA, AB, AC, A0, A1, BN, BZ, B7, ZZ, Z1.

Note: The user should not use any variable name which contains words with special meaning (or reserved words) in the BASIC language. For example, "CIF" cannot be used as a variable name, since it contains the BASIC keyword "IF".
A list of reserved words is in Appendix A.

Variable Types

There are four types of variables in the System 80 integer, single precision, double precision, and string variables. The first three types are used to store numeric values, whereas the last type is used only for character storage.

1. %: integer (whole numbers within the range -32769 to +32769)

   Example

   A% = - 30
   BB% = 8000

2. ! : single precision (6 significant digits)

   Example

   A ! = - 50.3
   D4! = .123456

3. # : double precision (16 significant digits)

   Example

   A# = 3.141592653589
   A2# = - 4567.8901234

4. $: string (maximum length : 255 characters)

   Example

   A$ = "SYSTEM 80"
   M2$ = "THE RESULT OF (A*B+15)/2.5 IS:"

Though A%, A!, A# , A$ all have the same variable name "A", their types are different, that is %, !, # , $; they are considered to be distinct variables by the computer.

Artithmetic Operators

Whenever any computation is needed in a program the arithmetic operators are used.

Example:

```
5  R = 6
10 A = R * 3.1416 * 2    :REM  COMPUTE THE CIRCUMFERENCE.
20 PRINT "THE CIRCUMFERENCE IS :";A
30 B = 3.1416 * R [ 2    :REM  COMPUTE THE AREA OF THE CIRCLE.
40 PRINT "THE AREA IS :";B
50 END
READY
>RUN
THE CIRCUMFERENCE IS : 37.6992
THE AREA IS : 113.098
```

The System 80 uses the general arithmetic symbols.

+ for addition, – for subtraction, * for multiplication, / for division, and [(the ESC key) for exponentiation.

For example, the result of $5 \times 12^{(1/3)}$ is equivalent to the result of `5 * 12 [ ( 1/3)` in System 80 (Note: you will find no [key on the keyboard, it is represented by the ESC key).

Relational Operators

Whenever a decision has to be made within a program, a relational operator is needed.
The acceptable operators are:

<	(less than)	< =	(less than or equal to)
>	(greater than)	> =	(greater than or equal to)
< >	(not equal)	=	(equal to)

Example:

```
120 IF A < B THEN PRINT "B IS GREATER THAN A. "
```

When the computer executes this statement, if the content of B is greater than the content of A (i.e. A < B is true), the sentence "B IS GREATER THAN A" will be printed on the screen. Otherwise the computer will just go to the next statement.

Logical Operators

AND, OR, and NOT are the only logical operators accepted by the System 80.

Example

```
10 IF A = 1 AND B = 5 GOTO 50
```

The computer branches to line 50 if A = 1 and B = 5, otherwise the computer goes to the next statement following line 10.

```
20 A = (B = 2) AND (C > 10)
```

A has the value of -1, if both B = 2 and C > 10 are true.
Otherwise A has the value of 0.

```
40 A = (D < 2) OR (E < 20)
```

A has the value of -1 if either D < 2 or E < 20 is true. When both D > = 2 and E > = 20 are true, then A has the value of 0.

```
70 A = NOT (F > 5)
```

A has the value of -1 if F < = 5.
Otherwise A has the value of 0.

String Operators.

In string operations, the relational operators are used to compare the precedence of two strings.

Note that the following operations are all true.

```
"B" < "C"                  ( THE CODE FOR B IS LESS THAN THE CODE FOR C )
"JOHN" > "JACK"            ( SAME REASON AS ABOVE. )
"STRING" = "STRING"
"LETTERS" <> "LETTERS "    ( SPACE ALSO COUNTS. )
A$ = "BO" + "AT"           ( A$ WILL HAVE THE VALUE : BOAT )
```

Order of Operations

Operations in the innermost level of parentheses are performed first, then evaluation proceeds to the next level, etc. Operations on the same level are performed according to the following precedence rules.

1.	Exponentiation	A [B
2.	Negation	-C
3.	Multiplication and Division	A*B, C/D
4.	Addition and Subtraction	C+D, E-F
5.	Relational Operators	A < B, "C" = "C", 15 < > 16
6.	Logical Operators	NOT, AND, OR

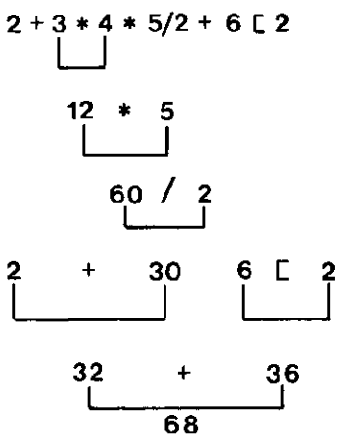
For example, we have a formula.

```
10 ANS = A + B * C * D / 2 + E [ 2
```

The computer will evaluate in the following sequence.

If A = 2
B = 3
C = 4
D = 5
E = 6

Then apply to the formula above



Therefore the answer should be 68.

CHAPTER 1

ACTIVE COMMANDS

Once the system is set up, with power on, the user should be in the Active Command level. The normal indication is the word "READY" followed by a " > " sign which appears on the next line at the upper left corner on the display (monitor or TV screen). For convenience we will call this indication the "ready message".

At this point, the user should hit the NEW LINE key before entering one of the following commands through the keyboard.

1. AUTO
2. CLEAR
3. CLOAD
4. CLOAD?
5. CONT
6. CSAVE
7. DELETE
8. EDIT
9. LIST
10. NEW
11. RUN
12. SYSTEM
13. TROFF
14. TRON
15. LLIST

We are going to discuss these commands separately. Please note that everything inside the brackets is optional. For example: AUTO (line number, increment) All the user has to do is type in the underlined portion:

```
AUTO 10. 5
```

or any numeric value to replace "line number" and "increment". In case the option is not taken, just type in

```
LIST
```

The computer will perform certain specified actions automatically. Notice: Every command should be followed by pressing the NEW LINE key.

1.1 AUTO (line number, increment)

This command automatically sets the line numbers before each source line is entered. The option permits the user to specify the beginning line number as well as the increment desired between lines. If the user only types in AUTO followed by the NEW LINE key, the beginning line number will be set at 10, with each increment of 10. The user may enter his program statement right after the line number.

Example

```
30 PRINT "THIS IS LINE 30."
```

Everytime the user hits the NEW LINE key, the computer will increment the line number. Until the BREAK key is hit, the AUTO command will remain in operation. (Note that whenever AUTO brings up a line that has been used previously, there will be an asterisk appear right next to the line number. If the user does not want to alter that line, hit the BREAK key to turn off the AUTO function).

Example

```
READY
>AUTO 1,2
1 LINE 1                    NEW LINE
3 LINE 3
5 LINE 5
7 LINE 7
9[BREAK]
READY
>AUTO 2,2
2 SECOND LINE               NEW LINE
4 FORTH LINE
6 SIXTH LINE
8[BREAK]

READY
>AUTO
10 LINE 10                  NEW LINE
20 LINE 20
30 LINE 30
40[BREAK]

READY
>AUTO 1,1
1*                          NEW LINE
2*
3*
4*
5*
```

1.2 CLEAR (number of bytes)

The command will clear a specific number of bytes for string storage. If the option is not used i.e. type in CLEAR followed by the NEW LINE key, the computer will reset all numeric variables to zero, and all string variables to null. When the option is taken, the command will perform, in addition to the first function, a second function: that is to clear a specified number of bytes for string storage. Note that when the user turns on the computer, a CLEAR 50 command is performed automatically.

Example

```
CLEAR 100
```

Reset all numeric variables to zero, and all string variables to null. Then clears 100 bytes of memory for string storage.

1.3 CLOAD (# – cassette number, "file name")

The command will load a specified program according to the "file name" to the computer from the appropriate cassette. Before using this command, the user should re-wind the cassette tape, check the cables and connectors (consult the user's manual), press the PLAY button on the cassette. If everything is ready, type in, for example CLOAD # –1, "A" then hit the NEW LINE key. The cassette will be turned on and starts searching until the file named "A" is found. If the file is found, a stable and a blinking asterisks will appear at the top right corner of the display to indicate loading is carrying out. Once the entire program has been loaded in the computer, the READY message will appear on the display.

Example

```
CLOAD #-1, "3"
```

Load from cassette No. 1 the file named "3".
Note that only the first character of the file name is used for CLOAD, CLOAD?, and CSAVE commands.

1.4 CLOAD? (file name)

This command will compare a specific program stored on cassette tape with the one in the computer's main memory. Usually, this command is used right after the CSAVE command which stores a program from the computer's main memory to a cassette. The CLOAD? command allows the user to examine whether the copying (CSAVE) operation is successful.

It is a good practice to include the file name in this command, since the computer will search for that file, or program, before comparison, starts. Otherwise the first file encountered on the cassette will be compared. During the operation, the program on tape and the program in memory are compared byte by byte. If any part does not match, the message "BAD" will be display. In this case, the user should repeat the CSAVE command again. Same as CLOAD Command, the cassette must be re-wound, cables and connectors checked, with the PLAY button on; prior hitting the NEW LINE key. (consult User's Manual for more details).

1.5 CONT

This command continues the program execution, at the point where the execution has been stopped by the BREAK key or a STOP statement within the program.

1.6 CSAVE#-cassette number, "file name"

This command stores the program in the computer's main memory onto cassette tape. Both the cassette number and the file name must be accompanied with this command. Any alphanumeric character other than double quotes (") will be acceptable as a file name. Again, before using the command, the cassette tape must be in a proper starting location (not overlapped with any useful program location). Check the cables and connectors, press the PLAY and REC buttons of the cassette at the same time, then start typing the command accordingly.

Example

```
CSAVE #-2, "C"
```

Saves a program with label "C" on cassette drive 2, from the main memory.
Warning: keep account of the locations of the saved programs on tape. Find an empty space for the new program to be loaded, unless you want to erase the old programs. Erased program are not recoverable. (Consult user's manual for more details).

1.7 DELETE line number (-line number)

This command will clear the memory location that contains the specified line(s).

Example

```
DELETE 5        Clear line 5
DELETE 7 - 10   Clear line 7 line, 10 and any line in between.
DELETE -12      Clear from the first line of the program, up to and including line 12.
DELETE .        Clear the line currently entered, or edited.
```

1.8 EDIT line number

This command will cause the computer to shift from the Active Command level to the Editing level. In the Editing level, the user is allowed to examine and modify the program statements in the main memory, by using a set of sub-commands. There must be a valid line number following the EDIT command, otherwise the command may not be accepted. Also see Chapter 2.

Example

```
EDIT 20
```

Turns the computer from Active Command level to Editing level – then examines line 20.

1.9 LIST (line number – line number)

This command will inform the computer to display any specified program lines stored in the main memory. If the option is not used, the computer will scroll the entire program onto the display. In order to pause and examine the text, the user should hit the SHIFT and @ keys simultaneously. The scrolling will continue by hitting any key.

Example

`LIST 3`	display line 3.
`LIST 10 - 20`	display line 10, line 20 and any line in between.
`LIST -50`	display from the first line up to and include line 50.
`LIST 20 -`	display line 20 and all following lines.
`LIST`	display the current line just entered or edited.
`LIST`	display all lines in the memory.

1.10 NEW

This command will clear all program lines; reset numeric variables to zero and string variables to null. It does not change the memory size previously set by the CLEAR command.

1.11 RUN (line number)

This command will instruct the computer to start executing (or RUN) the user's program stored in main memory. If a line number is not specified, the computer will start executing from the lowest line number. However, if a line number is provided, the computer will execute from the given line number to higher order lines. Note that an error will occur if an invalid line number is used.
Everytime a RUN is executed, a CLEAR command also executed automatically before it.

Example

`RUN 50`	start executing at line 50.
`RUN`	start executing at the lowest number line.

1.12 SYSTEM

This command turns the computer into the Monitor Mode. Within this mode, the user may load his own program or data file in machine code format.
To load an object file from tape, type in SYSTEM and NEW LINE; the "*?" symbol will be displayed. Then type in the file name. The tape will begin loading. When loading is completed, another "*?" will appear. Type in a slash "/" symbol followed by the entry point address (in decimal) where the user wants the execution to start.
If the user does not type in the entry address, execution will begin at the address specified by the object file.

1.13 TROFF

This command will turn off the Trace function. Usually follows the TRON command.

1.14 TRON

This command will turn on a Trace function that allows the user to keep track of the program flow for debugging and execution analysis. Everytime the computer executes a new program line, the line number will be displayed inside a pair of brackets.

Example

Consider the following program:

```
10 PRINT " **  PROGRAM 1 **"
20 A = 1
30 IF A = 3 THEN 70
40 PRINT A
50 A = A + 1
60 GOTO 30
70 PRINT " END PROGRAM 1. "
80 END
```

Type in

```
>TRON  NEW LINE
>RUN   NEW LINE

<10>   ** PROGRAM 1 **
<20><30><40> 1
<50><60><30><40> 2
<50><60><30><70>  END PROGRAM 1.
<80>
```

In order to pause execution before its natural end, the SHIFT and @ keys must be pressed simultaneously. To continue, just press any key.
To turn off the Trace function, enter TROFF. TRON and TROFF are available for use within user programs to check if a given line is executed.

Example

```
.
.
.
90 IF A = B THEN 160
100 TRON
110 A = B + C
120 TROFF
.
.
.
```

In this portion of a program, if A happens to be not equal to B, then line 110 should be executed. By using TRON and TROFF inside the program, the user can see precisely whether line 110 has been executed or not. The computer will display <110> <120> if these lines were executed. TRON and TROFF can be removed after a program is debugged.

1.15 LLIST

Lists a program onto the printer. This command functions in a very similar way as the LIST command. If the Line printer is not properly connected, the computer will enter a dead loop and waits to print the first character. This situation can only be resolved by turning the printer on or hitting the RESET button.

1.16 RE (starting line number, increment)

This command renumbers the BASIC program. After rearrangement of the line numbers, new statements can be inserted into the tightly packed program. In addition, it helps better program documentation. If the starting line number or increment value is not entered, it will be defaulted to 10.

Example:

RE, 5 NEWLINE — renumber program with starting line number equal to 10 and increment by 5.

RE NEWLINE — renumber program with both starting line number and increment value equal to 10.

CHAPTER 2

TEXT EDITING

The purpose of editing in the System 80 is to facilitate the user in modifying his programs. With the Editor, the user need not to type in the entire program every time he makes a programming mistake or typing error. The need for an editor becomes more critical when programs are long and complex.
Inside this chapter we discuss every editing function, including subcommands, that available for the System 80. A substantial amount of descriptive examples are presented with each command. Users are advised to try out each editing command before entering their first program into the system.

2.1 EDIT line number

This command shifts the computer from the Active Command level to the Editing level. The user must specify which line he wants to edit. If the line number is not provided, an FC error will occur (see Appendix B).

Example

```
EDIT 100 (allow to edit line 100)
EDIT.    (allow to edit the current line just entered.)
```

2.2 NEW LINE Key

Once the user presses the NEW LINE key while in the Edit mode, the computer will record all the changes made in that line, and return back to the Active Command level.

2.3 n Space-bar Key

In the Edit mode, pressing the space-bar will move the cursor one space to the right and display any character stored in the preceding position. The user may type in the value of n before hitting the Space-bar, then the cursor will move n spaces to the right side.
Suppose we have entered a line into the computer by the command:

```
>AUTO 100
100 IF A = B THEN 150 : A = A + 1 : GOTO 100
```

If the user wants to edit this line, he should type in EDIT 100 followed by the NEW LINE key, like the following:

```
>EDIT 100
```

then the display will become:

```
100_
```

By pressing the Space-bar 12 times, the cursor will move to the right side by 12 spaces. The display should look like:

```
100 IF A = B THE_
```

The user may also use the option to display more characters at once. That is, enter the number of cursor spaces desired, before hitting the Space-bar.

Example

Type in 8 followed by the Space-bar key:

```
100 IF A = B THE_
```

The display will become

```
100 IF A = B THEN 150 : _
```

If the user wants to display the next 20 positions, he may type 20 then the Space-bar again. The outcome should be:

```
100 IF A = B THEN 150 : A = A + 1 : GOTO 100_
```

2.4 n Backspace Key

This action will move the cursor back to the left by n spaces. If number n is not specified, the cursor only moves back one space at a time. Everything behind the cursor will disappear from the display; however, it is not erased from the memory.

Example

```
100 IF A = B THEN 150 . A = A + 1 : GOTO 100
```

Hit the Backspace key 5 times, the display will look like:

```
100 IF A = B THEN 150 : A = A + 1 : GOT_
```

Then type in 10 followed by Backspace key, the display will look like:

```
100 IF A = B THEN 150 : A = A_
```

After this sequence of operations, if the user hits the NEW LINE key, the display will look like:

```
>_
```

That means the computer has returned back to the Active Command level. If any further change is desired in line 100, the user must enter the Edit mode again.

2.5 SHIFT ESC Key

By pressing the SHIFT and ESC keys simultaneously, the computer will escape from any of the following Insert subcommands: H, I, X. After escaping from an Insert subcommand, the user remains in the Editing level, while the current cursor position is unchanged. Another way to escape from these Insert subcommands, is by pressing the NEW LINE key, which will shift the computer back to the Active Command level.

2.6 H Key

"H" represents Hack and Insert; that is to delete remainder of the line and to let the user insert material at the current cursor position.

Example

Consider this line:

```
100 IF A = B THEN 150 : A = A + 1 : GOTO 100
```

If the user wants to replace A = A + 1 by A = A + B, and to delete GOTO 100, he should first enter the Editing level, type in 25 followed by pressing the Space-bar (move 25 spaces from the beginning of the line). The display should look like:

```
100 IF A = B THEN 150 : A = A_
```

Now hit the H key, type in + B, then hit NEW LINE (back to the Active Command level). Or hit SHIFT and ESC simultaneously to return to Editing level, then hit L to display the entire line, as below:

```
100 IF A = B THEN 150 : A = A + B
100_
```

with anything not displayed being deleted.

2.7 [I] Key

"I" represents Insert, that is to allow insertion of characters starting at the current cursor position, without altering any other part of the line.

Example

We want to insert the statement "PRINT A" between "A = A + 1" and "GOTO 100" in line 100. Line 100 looks like:

```
100 IF A = B THEN 150 : A = A + 1 : GOTO 100
```

By using the EDIT mode and the [Space-bar] Move the cursor to:

```
100 IF A = B THEN 150 : A = A + 1 : _
```

Now hit the I key, type in "PRINT A :", then press the [SHIFT] and [ESC] keys to escape from the subcommand level. At this point we can type in [L] to list the current line. And the display should look like:

```
100 IF A = B THEN 150 : A = A + 1 : PRINT A : GOTO 100
100 _
```

or we can hit the [NEW LINE] key to return to the Active Command level.

2.8 [X] Key

"X" represents Insert at End of Line. The command moves the cursor position to the end of the line, and shifts the computer into the Insert subcommand. The user can insert new materials at the end of the line, or delete part of the existing line by using the [Backspace] key.

Example

Get into the Edit mode

```
>EDIT 100
100 _
```

Type in X without hitting NEW LINE key. The line displayed should be

```
100 IF A = B THEN 150 : A = A + 1 : PRINT A : GOTO 100
100 _
```

At this point, the user may add some new material, or delete part of the existing line, before hitting SHIFT and ESC

2.9 L Key

"L" represents List line. While the computer is in the Editing level, and is not currently executing one of the subcommands H, I, X ,the L command will list the remaining part of the line onto the display.

Example

```
>EDIT 100
100 _
```

Hit L (without hitting NEW LINE), the display should be:

```
100 IF A = B THEN 150 . A = A + 1 . PRINT A . GOTO 100
100 _
```

The second line allows the user to do editing, while referencing the first line.

2.10 A Key

"A" represents Cancel and Restart. In the Editing level this command moves the cursor back to the beginning of the line, cancels all editing changes previously made on that line, and restores the former content of the line.

2.11 E Key

This command shifts the computer from Editing level back to the Active Command level, and saves all the changes previously made. Make sure the computer is not executing any subcommand before entering E.

2.12 [Q] Key

This command shifts the computer from Editing level back to the Active Command level, but cancel all the changes made in the current edit mode. Just type in Q to cancel the changes made and return to the Active Command level.

2.13 n [D] Key

"D" represents delete; the command will delete n numbers of characters right after the current cursor position. The deleted characters will be enclosed in exclamation marks "!" to show you which characters are being affected.

Example

Consider the following line:

```
100 IF A = B THEN 150 : A = A + 1 . PRINT A : GOTO 100
```

We first enter into the Editing level, move the cursor to the following position:

```
100 IF A = B THEN 150 : A = A + 1
```

Now type in 15D (to delete 15 characters); the display should look like:

```
100 IF A = B THEN 150 : A = A + 1! : PRINT A : GO!
```

Then use L to list the entire line, the display should become:

```
100 IF A = B THEN 150 : A = A + 1! : PRINT A : GO!TO 100
100 _
```

List Again:

```
100 IF A = B THEN 150 : A = A + 1TO 100
100 _
```

Now use the X key and the Backspace key to delete "TO 100"; the final outcome should be:

```
100 IF A = B THEN 150 : A = A + 1
```

2.14 n [C] Key

"C" represents change; the command allows the user to change n number of characters right after the current cursor position. If the number n is not specified, the computer assumes the user only wants to change a single character.

Example

Consider the line

```
100 IF A = B THEN 150 : A = A + 1
```

If the user wants to change 150 to 230, he should enter the Edit mode and move the cursor to the following position:

```
100 IF A = B THEN
```

Now type in 2C (change the next 2 characters), followed by 23 (new data), then hit the [SHIFT] and [ESC] keys. List the line by hitting [L]:

```
100 IF A = B THEN 230 : A = A + 1
100 _
```

2.15 n [S] c

The command searches for the n th occurrance of the character c on that line and moves the cursor to that position. If the n value is not provided, the computer will search for the first occurrance of the character specified and stop the cursor there. In case the specified character is not found, the cursor will move to the end of the line. As usual, the computer will start searching from the current cursor position toward the right end of the line.
Consider the following example:

```
100 IF A = B THEN 230 : A = A + 1
```

After entering the Edit mode, the display should look like:

```
100 _
```

Now type in 2S = , to inform the computer to search for the second occurrance of the equal sign "=", and the final display should be

```
100 IF A = B THEN 230 : A _
```

Now, the user may enter one of the subcommands at the current cursor position. For example:
Type in H (hack and insert) followed by "= A + 2" (new data).
Then the line will become:

```
100 IF A = B THEN 230 : A = A + 2 _
```

2.16 n [K] c

The command will delete all characters up to the n th occurrance of character C, and move the cursor to that position. Consider the following example:

```
100 IF A = B THEN 230 : A = A + 2
```

Enter into the Edit mode:

```
100_
```

Now type in 1K:, to inform the computer to search for the first occurrance of the colon ":" symbol, then delete everything in front of it on that line.
The display should become

```
100 !IF A = B THEN 230 !
```

The ":" should also be deleted so type in D, the display will become:

```
100 !IF A = B THEN 230 !!:!
```

Then hit the L key to list the line on the display. The line should look like

```
100 A = A + 2
100_
```

CHAPTER 3

BASIC PROGRAMMING STATEMENTS

In this chapter, we are going to discuss the program statements in our BASIC language. The first part of this chapter covers all the Input-Output statements available for the computer to communicate with the outside world; essentially through the keyboard and video display, as well as storing to and retrieving from cassette tapes.

The second part of this chapter concerns various functions of all the programming statements in BASIC which are acceptable to the System 80. Since it is a very large set of statements, and each statement has its own unique and characteristics in programming, the users are advised to study each statement with the help of the examples provided.

INPUT - OUTPUT STATEMENTS:

3.1 PRINT item list

Prints an item or a list of items on the display. Item may be any of the following:

a) Numeric constants (numbers such as 0, 36872, 0.2, - 34)
b) Numeric variables (names respresenting numeric values, such as X, Y, Z, etc.)
c) String constants (characters enclosed in quotes, such as "HOME COMPUTER", "3003", etc.)
d) String variables (names representing string or character values, such as A$, B$, etc.)
e) Expressions (a sequence of any combination of the above, such as (X + 10)/Y, "BALL" + "PEN", etc.)

Items in the item list may be separated by commas or semi-colons. If commas are used, the cursor automatically advances to the next printing zone before printing the next item. If semi-colons are used, no space is inserted between alphabetic items before printing on the display, but one space is inserted before each numeric item.

Example

```
10 N = 25 + 7
20 PRINT "25 + 7 IS EQUAL TO ";N
30 END
```

```
READY
>RUN

25 + 7 IS EQUAL TO  32
```

Example

```
10 H$ = "HOME "
20 C$ = "COMPUTER"
30 PRINT "TRY OUR ";H$;C$
40 END

READY
>RUN

TRY OUR HOME COMPUTER
```

When commas are used to separate items, 4 columns are acceptable per line. Each column consists of a maximum of 16 characters. Any string beyond this bound will be printed on the next line.

Example

```
10 PRINT "COLUMN 1","COLUMN 2","COLUMN 3","COLUMN 4","COLUMN 5"
20 END

READY
>RUN

COLUMN 1        COLUMN 2        COLUMN 3        COLUMN 4

COLUMN 5
```

If two or more commas are applied together, each comma will still occupy 16 characters. (Blank spaces).

Example

```
10 PRINT "COLUMN 1",,"COLUMN 2"
20 END

READY
>RUN

COLUMN 1                        COLUMN 2
```

Note the following examples:

```
10 PRINT "LINE ONE"
20 PRINT "LINE TWO"
30 END

READY
>RUN

LINE ONE
LINE TWO

10 PRINT "LINE ONE",
20 PRINT "LINE TWO"
30 END

READY
>RUN

LINE ONE        LINE TWO
```

3.2 PRINT@ location, item list

This statement prints out items in the item list at the screen location specified. The "@" sign must follow PRINT immediately, and the location specified must be a number of value from 0 to 1023. For more details on the display map, please refer to Appendix E.

Example

```
20 PRINT @100, "LOC 100"
```

If the user constructs a PRINT@ statement to print on the bottom line of the diplay, there will be an automatic line-feed, causing everything displayed to move up one line. To suppress this action, add a semi-colon at the end of the statement.

Example

```
10 PRINT @ 999 , "BOTTOM LINE";
```

3.3 PRINT TAB(expression)

Allows the user to print at any specified cursor position within a line. More than one TAB in a PRINT statement is acceptable. However, the value in the expression must be between 0 and255 inclusive.

Example

```
10 PRINT TAB(10) "POSITION 10" TAB(30) "POSITION 30"
20 END

READY
>RUN
       POSITION 10          POSITION 30

10 N = 4
20 PRINT TAB(N) "POS. ";N TAB(N+10) "POS. ";N+10 TAB(N+20) "POS. ";N+20
30 END
READY
>RUN
  POS.  4     POS.  14    POS.  24
```

3.4 PRINT USING format, item list

This statement allows the user to print the data with a pre-defined format. The data can be numeric or string values.

The format and item list in PRINT USING statement can be expressed as variables or constants. The statement prints the item list according to the format specified.

The following specifiers may be used in the format field.

\# This sign represents the proper position of each digit in the item list (for numeric value). The number of # signs used forms the format desired. If the format field is greater than the numeric value (in the item list), the unused field positions to the left of the number will be displayed as spaces and those to the right of the decimal point will be displayed as zeros.

The decimal point can be placed anywhere in the format field established by the # signs. Rounding off will take place if the digits to the right of the decimal point are suppressed.

The comma – When it is placed at any position between the first digit and the decimal point, a comma will be displayed to the right of every three digits.

Let us consider the following examples:

```
10 INPUT "ENTER FORMAT ";F$
20 IF F$ = "STOP" END
30 INPUT "ENTER A NUMBER ";N
40 PRINT USING F$;N
50 GOTO 10
```

This program requests inputs for the format field and item list (in this case with numeric value). The program will stop only if the user inputs the word "STOP" as the value for F$.

Now try to run this program.

```
READY
>RUN
ENTER FORMAT ?##.##
ENTER A NUMBER ? 12.34
12.34
ENTER FORMAT ?###.##
ENTER A NUMBER ? 12.34
 12.34
ENTER FORMAT ?##.##
ENTER A NUMBER ? 123.45
%123.45
ENTER FORMAT ?STOP
```

The % sign will be automatically printed out if the field is not large enough to contain the number of digits found in the numeric value. The entire number to the left of the decimal point will be displayed after the % sign.

Let us run tne program again.

```
READY
>RUN
ENTER FORMAT ?##.##
ENTER A NUMBER ? 12.345
12.35
ENTER FORMAT ?STOP
```

Since only two decimal places were specified, the numeric value will be rounded-off before displaying to the screen.

(i) ** Two asterisks placed at the beginning of the format field will cause all unused positions to the left of the decimal point to be filled with asterisks. The two asterisks will establish two more positions in the field.

(ii) $$ Two dollar signs placed at the beginning of the field will act as a floating dollar sign. That is: A dollar sign will occupy the first position preceding the number.

(iii) **$ Combines the effects of ** and $$. Any empty position to the left of the number will be filled by the * sign and the $ sign will also occupy the first position preceding the number.
Let us use the same example as before:

```
READY
>RUN

ENTER FORMAT ?****. ##
ENTER A NUMBER ? 12.3
**12.30
ENTER FORMAT ?$$###. ##
ENTER A NUMBER ? 12.34
 $12.34
ENTER FORMAT ?**$###. ##
ENTER A NUMBER ? 12.34
***$12.34
ENTER FORMAT ?STOP
```

(iv) + When a "+" sign is placed at the beginning or at the end of the format field, the computer will print a + sign for a positive number or a – sign for a negative number at the specific position accordingly.

(v) – When a "–" sign is placed at the end of the format field, it will cause a negative sign to be printed after any negative number, and will display as a blank for positive numbers.

Examples (using the same program as above)

```
READY
>RUN

ENTER FORMAT ?#####,#
ENTER A NUMBER ? 12345.6
12,346
```

```
ENTER FORMAT ?+##. ##
ENTER A NUMBER ? 12.34
+12.34
ENTER FORMAT ?+##. ##
ENTER A NUMBER ?-12.34
-12.34
ENTER FORMAT ?##. ##+
ENTER A NUMBER ?-12.34
12.34-
ENTER FORMAT ?##. ##-
ENTER A NUMBER ? 12.34
12.34
ENTER FORMAT ?##. ###
ENTER A NUMBER ? 123456
%123456.000
ENTER FORMAT ?STOP
```

(vi) % space %

To define a string field of more than one character. The length of the format field will be 2 plus the number of spaces between the percentage signs. An exclamation mark (!) informs the computer to use only the first character of the current string value.

Consider the following program example:

```
10 INPUT "ENTER FORMAT ";F$
20 IF F$ = "STOP" END
30 INPUT "ENTER A STRING ";C$
40 PRINT USING F$;C$
50 GOTO 10
```

This program performs similarily to the one we just used. The only difference is that, the user has to input a string value instead of a numeric value for the second data entry. This is, the variable C$.

Now let us run the program and test its function.

```
READY
>RUN
ENTER FORMAT ?!
ENTER A STRING ?ABCDE
A
ENTER FORMAT ?% %
ENTER A STRING ?ABCDE
ABC
ENTER FORMAT ?%   %
ENTER A STRING ?ABCDEF
ABCDE
ENTER FORMAT ?STOP
```

(vii) ! By using the ! sign, we can also concatenate, or join strings together.

Example

```
10 INPUT "ENTER THREE STRINGS ";A$,B$,C$
20 PRINT "THE RESULT IS :";:PRINT USING "!";A$;B$;C$
30 END
```

Now, run the program.

```
READY
>RUN

ENTER THREE STRINGS ?ABC,XYZ,IJK
THE RESULT IS :AXI

ENTER THREE STRINGS ?A,COMPUTER,PROGRAM
THE RESULT IS :ACP
```

By using more than one "!" signs, the first letter of each string will be printed with spaces inserted corresponding to the spaces inserted between the "!" signs.

Try to follow this example:

```
10 INPUT "ENTER THREE STRINGS "; A$, B$, C$
20 PRINT "THE RESULT IS :"; :PRINT USING "! ! !"; A$; B$; C$
30 END

READY
>RUN

ENTER THREE STRINGS ?XYZ, FGH, ABC
THE RESULT IS :X F A


ENTER THREE STRINGS ?A, COMPUTER, PROGRAM
THE RESULT IS :A C P
```

3.5 INPUT item list

This statement causes the computer to suspend execution of a program and wait until the user has input the specified number and type of values through the keyboard. Input values can be string or numeric according to the variable type. The items (if more than one) in the list must be separated by commas.

Example

```
10 INPUT A$, B$, A, B
```

This statement permits the user to input two string values, followed by two numeric values. The input sequence must be consistent. When the computer executes this statement, it sends a signal onto the display:

```
?
```

And waits for the inputs. The user may enter all four values at once (separated by commas). In this case, the inputs could be as follow:
orange, apple, 59, 47 [NEW LINE]

The computer then assigns the values accordingly:

```
A$ = "ORANGE"
B$ = "APPLE"
A = 59
B = 47
```

The other way to input those values would be by entering the items on separate lines. In this way, the computer will remind the user to input the next data for the remaining variables by displaying:

```
??
```

Until all variables are set, the computer then advances to the next statement. Input must be compatible to the variable type specified. In other words, the user should not input a string value to a numeric variable. If such an invalid entry occurs, the computer will send the message:

```
? REDO

?
```

Indicating the input does not match with the current variable type. However, the computer gives the user a second chance to input the correct data starting with the first value expected by the INPUT statement.

Example

```
10 INPUT A$,A
20 PRINT A$,A
30 END

READY
>RUN

? STRING, 10
STRING          10
```

```
READY
>RUN
? THIS IS A STRING, 13.5
THIS IS A STRING                13.5

READY
>RUN

? ABCD, IJK
? REDO
? ABCDE
?? 25
ABCDE      25
```

If an input string consists of blanks, the entire string must be enclosed by quotes.

In order to provide a clearer indication to the operator, the user may include a "prompting message" in the INPUT statement. This helps to input correct data type to each variable. The prompting message must immediately follow INPUT, enclosed in quotes, and followed by a semi-colon.

Example

```
100 INPUT "INPUT ITEM NAME AND QUANTITY ";N$,Q
```

```
READY
>RUN
INPUT ITEM NAME AND QUANTITY ?
```

3.6 DATA item list

This statement allows the user to store data inside the program and to access them through READ statements. The item list will be accessed by the computer sequentially, starting with the first item in the first DATA statement, and ending with the last item in the last DATA statement. Each item in the item list may be a string or a simple numeric value. Just like entering data from the keyboard, any string value consists of blanks, colons, commas, must be enclosed in a pair of quotes.
The order of values in a DATA statement must match up with the variable types in the READ statements. DATA statements may appear anywhere in a program.

Example

```
10 READ A$, B$, C, D
20 PRINT A$, B$, C, D
30 DATA "CHARACTERS", "A LONG SENTENCE"
40 DATA 20, 137.54
50 END


READY
>RUN
CHARACTERS      A LONG SENTENCE  20              137.54
```

3.7 READ item list

This statement instructs the computer to read in a value from a DATA statement and assign that value to the specified variable. The values in the DATA statement will be read sequentially by the READ statement. After all the items in the first DATA statement have been read, the next READ statement encountered will access the second DATA statement for the next variable. If there is no more value in the DATA statement available for a READ statement an Out-of-Data error will occur.

Consider the following example:

```
10 READ C$
20 IF C$ = "EOF" GOTO 60
30 READ Q
40 PRINT C$, Q
50 GOTO 10
60 PRINT:PRINT "END OF LIST. ":END
70 DATA BOOKS, 4, PENCILS, 12
80 DATA BALL PENS, 5, COMPASSES, 2
90 DATA GLASSES, 5, EOF


READY
>RUN

BOOKS           4
PENCILS         12
BALL PENS       5
COMPASSES       2
GLASSES         5

END OF LIST.
```

3.8 RESTORE

This statement allows the next READ statement to access the first item in the first DATA statement, and the subsequent items.

Example

```
10 READ A$, A
20 PRINT A$, A
30 RESTORE
40 READ B$, B
50 PRINT A$, A, B$, B
60 DATA "JOHN WHITE", 25, "JOE HUDSON", 32, "BILL ADAMS", 30
70 END
```

```
READY
>RUN

JOHN WHITE        25
JOHN WHITE        25              JOHN WHITE        25
```

This program shows that the RESTORE statement not only allows the READ statement to access the first item in the first DATA statement, but also it has no effect on the previous assignments.

3.9 PRINT # – cassette number, item list

This statement prints the values of the specified variables onto cassette tape. The recorder must be properly set in record mode before executing this statement. For more detail, please consult the User's Manual. As the System 80 can control up to two cassette drives, the user should specify which drive is intended.

Example

```
10 A$ = "BEGIN TAPE"
20 B = 3.1416
30 C = 50
40 D$ = "DATA"
50 PRINT #-1, A$, B, C, D$, "END OF FILE"
60 END
```

This program assigns various data to variables A$, B, C, and D$ respectively, then PRINT these data on tape through cassette drive No. 1. Note that the string constant "END OF FILE", can be printed on tape as well as variables. Once the data are stored on tape, the user may input these data into the computer again, just like playing music tapes with a cassette. Please note that the INPUT statement must be identical to the PRINT statement in terms of number and types of variables. However, the variable names may be different in any case.

Important:

The total number of characters respresented in all the variables mentioned in the "item list" must not exceed 255; otherwise anything after the 255th character with be truncated or lost.

Example

```
10 PRINT #-1, A$, B$, C$, D$, E$
```

If the total number of characters in A$, B$, C$, D$, are 250 and E$ has a length of 35 characters, then E$ will not be saved on tape. And if the user tries to INPUT the value of E, an Out-of-Data error will occur.

3.10 INPUT # – cassette number, item list

This statement tells the computer to input the specified number of values stored on the cassette tape and to assign them to the variables. The user must specify the cassette drive number from which data is expected.

Example

```
10 INPUT #-1, A$, B, C, D$
```

This statement inputs data from cassette drive number 1. The first value is assigned to A$, the second value to B, etc. The cassette deck must be in PLAY mode. Once the computer executes this statement, the cassette drive will be turned on, and when the input has finished, the cassette drive will be turned off before the computer goes to the next statement.

If a string is encounted when a numeric value is expected by the INPUT statement, a bad file data error will occur. An Out-of-Data error will also occur if there is not enough data items on the tape for all the variables in an INPUT statement.

3.11 DEFINT letter range

Variable names that begin with letters specified within the letter range, will be treated and stored as integers. However, a type declaration character (refer to the Introduction) can over-ride this type definition. Defining a variable name as an integer not only saves memory space, but also saves computer time, because integer calculation is faster than single or double precision calculation. Note that integers can only take on values between – 32768 + 32767 inclusive.

Example

```
10 DEFINT X,Y,Z
```

After the computer has executed line 10, all variables beginning with the letters X, Y, or Z will be treated as integers. Therefore, X2, X3, YA, YB, ZI, ZJ will become integer variables. Except that X1 # , X2 # , YB #, will be still double precision variables, because type declaration characters always over-ride DEF statements.

Example

```
10 DEFINT A - D
```

Causes variables beginning with letter A, B, C, or D to be integer variables.
Note that DEFINT can be placed anywhere in a program, but it may change the meaning of variable references without type declaration characters. Therefore, it is normally placed at the beginning of a program.

3.12 DEFSNG letter range

Variable names that begin with those letters specified within the letter range, will be treated and stored as single precision variables. However, a type declaration character can over-ride this type definition.
Single precision variables and constants are stored with 7 digits of precision and printed out with 6 digits of precision. All numeric variables are assumed to be single precision unless otherwise specified. The DEFSNG statement is primarily used to re-define variables which have previously been defined as double precision or integer.

Example

```
10 DEFSNG A-D, Y
```

Causes variables beginning with the letter A through D, or Y to become single precision. However, A# would still be a double precision variable and Y% still be an integer variable.

3.13 DEFDBL letter range

Variable names that begin with those letters specified within the letter range, will be treated and stored as double precision. However, a type declaration character can over-ride this type definition. Double precision allows 17 digits of precision, while only 16 digits are displayed when a double precision variable is printed.

Example

```
10 DEFDBL M-P, G
```

Causes variables beginning with one of the letters M through P, or G to become double precision.

3.14 DEFSTR letter range

Variables that begin with those letters specified within the letter range, will be treated and stored as string.
However, a type declaration character can over-ride this type definition. Each string can store up to 255 characters, if there is enough string storage space cleared.

Example

```
10 DEFSTR A-D
```

Causes variables beginning with any letter A through D to be string variables, unless a type declaration character is added. Therefore, after the execution of line 10, the assignment B3 = "A STRING" is valid.

3.15 CLEAR n

This statement sets all variables to zero. If number n is specified, the computer sets n bytes of space for string storage. Everytime when the System 80 is turned on, 50 bytes of space are automatically cleared and reserved for strings.
The CLEAR statement becomes critical during program execution, because an Out of String Space error will occur, if the amount of string storage cleared is less than the greatest number of characters stored in string variables.

Example

```
10 CLEAR 1000
```

Clear 1000 bytes of memory space for string storage.

3.16 DIM name (dim 1, dim 2 . . . . . dim n)

The statement defines the variable name to be an array or list of arrays. The number of elements in each dimension may be specified through dim 1, dim 2, etc. If dim n is not specified, 11 elements in each dimension is assumed in each array. The number of dimensions is limited only by the memory size available.

Example

```
10 DIM A(5), B(3,4), C(2,3,3)
```

This statement defines the one dimensional array with 6 elements (from 0 to 5); the two dimensional array B with 20 elements (4 x 5); the three dimensional array C with 48 elements (3 x 4 x 4).
DIM statements may be placed anywhere in a program, and the number of subscripts may be an integer or an expression.

Example

```
10 INPUT "NUMBER OF TIMES ";N
20 DIM A(N+2,4)
```

The number of elements in array A may vary according to N.
To re-dimension an array, the user must use a CLEAR statement either with or without the argument n. Otherwise an error will occur.

Example

```
10 X(2) = 13.6
20 PRINT "THE SECOND ELEMENT IS :";X(2)
30 DIM X(15)
40 PRINT X(2)
50 END


READY
>RUN

? DD ERROR IN 30
```

3.17 LET variable = expression

This statement is used to assign a value to a variable. The word LET is not required in assignment statements by the System 80 BASIC interpreter. However, the user may use the word LET in order to make the program compatible with other systems.

Example

```
10 LET A = 5.67
20 B% = 20
30 S$ = "CHARACTERS"
40 LET D% = D% + 1
50 PRINT A, B%, S$, D%
60 END

READY
>RUN
5.67            20              CHARACTERS      1
```

In all the assignments above, the variable on the left of the equal sign is assigned with the value of the constant or expression on the right side. All these statements are acceptable.

3.18 END

This statement causes a normal termination of program execution. The END statement is primarily used to cause execution to terminate at some point other than the logical end of the program.

Example

```
5  B = 3: C = 14
10 A = C + B
20 GOSUB 70
30 D = X + Y
40 PRINT "THE RESULTS ARE :";
50 PRINT A, D
60 END
```

```
70 X = 50
80 Y = A * X
90 RETURN

THE RESULTS ARE : 17            900
```

The END statement in line 60 prevents the computer from executing into line 70. Therefore the subroutine that starts at line 70 can be accessed only by line 20.

3.19 STOP

This statement is essentially a debugging aid. It sets a break point in a program during execution, and allows the user to examine or modify variable values. A message will be printed out as "BREAK IN line number" once the computer executes the STOP statement. The Active Command CONT can then be used to re-start execution at the point where it breaks.

Example

```
5 INPUT B,C
10 A = B + C
20 STOP
30 X = (A + D)/0.74
40 IF X < 0 GOTO 70
50 PRINT A,B,C
60 PRINT X
70 END

READY
>RUN
?2,4
BREAK IN 20
READY
>PRINT A
 6
READY
>CONT

6               2               4
8.10811
```

The STOP statement allows the user to examine the value of A before line 30.

3.20 GOTO line number

This statement transfers program control to the specified line number. If used independently, an unconditional branch will result. However, test statements may precede the GOTO statement to create a conditional branch.

Example

```
10 A = 10
20 B = 45
30 C = A + B
40 C = C * 3.4
50 GOTO 100
60 .
70 .
80 .
90 .
100 PRINT "A =";A, "B=";B, "C=";C
110 END
READY
>RUN
A = 10          B= 45           C= 187
```

When line 50 is executed, control will unconditionally jump to line 100.

Example

```
10 IF A = 2 GOTO 120
```

When line 10 is under execution, if A equals to 2 then control will jump to line 120, otherwise it will just go to the next statement.
The user may use GOTO in the Active Command level as an alternative to RUN command. GOTO line number causes execution to begin at the specified line number, but without the automatic CLEAR.

3.21 GOSUB line number

Transfers program control to the specified line number where a subroutine starts. Only if the computer encounters a RETURN statement, it will then jump back to the statement that immediately follows the GOSUB. Just like GOTO, GOSUB may be preceded by a test statement, such as:
IF A = B THEN GOSUB 100

Example

```
10 PRINT "MAIN PROGRAM."
20 GOSUB 50
30 PRINT "END OF PROGRAM."
40 END
50 PRINT "SUBROUTINE."
60 RETURN
READY
>RUN

MAIN PROGRAM.
SUBROUTINE.
END OF PROGRAM.
```

3.22 RETURN

This statement ends a subroutine and returns control to the statement that immediately follows the GOSUB. An error will occur if RETURN is encountered without execution of a matching GOSUB.

3.23 ON n GOTO line number list

This statement allows multi branching to the line numbers specified according to the value of n. The general format for ON n GOTO is:

ON expression GOTO 1st line number, 2nd line number, . . . , mth line number.

The value of the expression must be between 0 and 255 inclusive.

When ON-GOTO statement is executed, first, the expression is evaluated and the integer portion, that is INT (expression) is obtained. Then the computer assigns this integer to N, and counts over to the Mth element in the line number list, and then branches to the line number specified by that element. If N is greater than the available line number M, the control fall through to the next statement in the program.
If the expression or number is less than zero, an error will occur.
The line number list may contain any number of items.

Example

```
10 INPUT "ENTER COMMAND ";C
20 ON C GOTO 100,120,130,150,130
30 PRINT "END OF PROGRAM.":END
100 PRINT "THIS IS LINE 100":GOTO 10
120 PRINT "THIS IS LINE 120":GOTO 10
130 PRINT "THIS IS LINE 130":GOTO 10
150 PRINT "THIS IS LINE 150":GOTO 10
READY
>RUN
ENTER COMMAND ? 5
THIS IS LINE 130
ENTER COMMAND ? 4
THIS IS LINE 150
ENTER COMMAND ? 1
THIS IS LINE 100
ENTER COMMAND ? 2
THIS IS LINE 120
ENTER COMMAND ? 3
THIS IS LINE 130
ENTER COMMAND ? 0
END OF PROGRAM.
READY
>RUN
ENTER COMMAND ? 4
THIS IS LINE 150
ENTER COMMAND ? 6
END OF PROGRAM.
```

The ON-GOTO statement is a more elegant way of achieving the some result than the equivalent IF-GOTO statements:

```
10 IF C = 1 GOTO 100
20 IF C = 2 GOTO 120
30 IF C = 3 GOTO 130
40 IF C = 4 GOTO 150
50 IF C = 5 GOTO 130
60 IF C < 1 OR C > 5 GOTO 70  :REM  GO TO THE NEXT STATEMENT.
```

3.24 ON n GOSUB line number list

Works like ON n GOTO, except control branches to one of the subroutines specified by the line numbers in the line number list.

Example

```
10 PRINT "  **  FUNCTION SUBROUTINES  **"
20 PRINT "    1.  FUNCTION A"
30 PRINT "    2.  FUNCTION B"
40 PRINT "    3.  FUNCTION C"
50 INPUT "ENTER 1, 2, OR 3 ";N
60 ON N GOSUB 150,100,250
70 END
100 PRINT "THIS IS FUNCTION B" : RETURN
150 PRINT "THIS IS FUNCTION A" : RETURN
250 PRINT "THIS IS FUNCTION C" : RETURN
READY
>RUN
  **  FUNCTION SUBROUTINES  **
    1.  FUNCTION A
    2.  FUNCTION B
    3.  FUNCTION C
ENTER 1, 2, OR 3 ? 2
THIS IS FUNCTION B
READY
>RUN
  **  FUNCTION SUBROUTINES  **
    1.  FUNCTION A
    2.  FUNCTION B
    3.  FUNCTION C
ENTER 1, 2, OR 3 ? 1
THIS IS FUNCTION A
```

3.25 FOR name = expression TO expression STEP expression

NEXT name

These statements form an iterative loop so that a sequence of program statements may be executed over a specified number of times.
The general form is:
FOR counter = initial value TO final value STEP increment.
*
*
*
Statements
*
*
*
NEXT counter

In the FOR statement, initial value,final value and increment can be constants, variables or expressions. The first time the FOR statement is executed, these three are evaluated and the values are saved; if these values are changed inside the loop, they will have no effect on the loop's operation. However, the counter value must not be changed or the loop will not operate normally.

The FOR-NEXT loop works as follows: the first time the FOR statement is executed, the counter is set to the "initial value". Execution proceeds until a NEXT statement is encountered. At this point, the counter is incremented by the amount specified in the STEP increment. If STEP increment is not used, an increment of 1 is assumed. However, if the increment has a negative value, then the counter is actually decremented.

The counter is then compared with the final value specified in the FOR statement. If the counter is greater than the final value, the loop is completed and execution continues with the statement following the next statement.(if increment was a negative number, loop ends when counter is less than the final value.)

If the counter has not yet exceeded the final value, control passes back to the first statement after the FOR statement.

Example

```
10 FOR K = 0 TO 1 STEP 0.3
20 PRINT "THE VALUE OF K :";K
30 NEXT K
40 END
READY
>RUN

THE VALUE OF K : 0
THE VALUE OF K : .3
THE VALUE OF K : .6
THE VALUE OF K : .9
```

When K = 1.2, it is greater than the final value 1, therefore the loop ends without ever printing 1.2.

Example

```
10 FOR N = 5 TO 0
20 PRINT "THE VALUE OF N :";N
30 NEXT N
40 END

READY
>RUN

THE VALUE OF N : 5

10 FOR N = 5 TO 0 STEP -1
20 PRINT "THE VALUE OF N :";N
30 NEXT N
40 END

READY
>RUN

 THE VALUE OF N : 5
 THE VALUE OF N : 4
 THE VALUE OF N : 3
 THE VALUE OF N : 2
 THE VALUE OF N : 1
 THE VALUE OF N : 0
```

Since no STEP was specified, so STEP 1 is assumed. N is incremented the first time, and its value becomes 6. Because 6 is greater than the final value 0, the loop ends. This is remedied by adding STEP-1, as you can see.

Example

```
10 FOR A = 0 TO 3
20 PRINT "THE VALUE OF A :";A
30 NEXT
40 END
READY
>RUN
THE VALUE OF A : 0
THE VALUE OF A : 1
THE VALUE OF A : 2
THE VALUE OF A : 3
```

Note here that instead of using NEXT A in line 30, you may simply write NEXT. However, this can lead to trouble if you have nested FOR-NEXT, loops.

Here is an example of nested loops, showing how it is advisable to identify the counter variable in each NEXT statement:

```
10 I = 1
20 J = 2
30 K = 3
40 FOR N = I + 1 TO J + 1
50     PRINT "FIRST LOOP"
60     FOR M = I TO K
70         PRINT "  SECOND LOOP"
80     NEXT M
90 NEXT N
100 END
READY
>RUN
FIRST LOOP
  SECOND LOOP
  SECOND LOOP
  SECOND LOOP
FIRST LOOP
  SECOND LOOP
  SECOND LOOP
  SECOND LOOP
```

3.26 ERROR Code

This statement is used for testing an ON ERROR GOTO routine. When the ERROR code statement is encountered, the computer will proceed exactly as if that kind of error has occurred.

Example

```
30 ERROR 1

?NF ERROR IN 30
```

For the definition of each error code, please refer to Appendix B.

3.27 ON ERROR GOTO line number

This statement allows the user to set up an error-trapping routine to recover a program from an error and to continue, without any break in execution. Without this statement, the computer will stop execution and print out an error message, once it encounters any kind of error in the user's program. Normally, the user has a particular type of error in mind when an ON ERROR GOTO statement is used.
For example, suppose that a program performs some division operations and the user has not ruled out the possibility of division by zero. The user could write a routine to handle a division-by-zero error, and then use ON ERROR GOTO to branch to that routine when such an error occurs.

Example

```
5 B = 15 : C = 0
10 ON ERROR GOTO 120
20 A = B/C
30 PRINT A, B, C
40 END
120 PRINT "DIVIDED BY ZERO !!"
130 END
READY
>RUN
DIVIDED BY ZERO !!
```

In this example, C has a value of zero, so a divide-by-zero error will occur when the computer attempts to execute line 20. But because of line 10, the computer will simply ignore line 20 and branch to the error-handling routine beginning at line 120. Please note that the ON ERROR GOTO statement must be executed before the error occurs, otherwise it has no effect. Note also that the error handling routine must be terminated by a RESUME statement.

3.28 RESUME line number

This statement terminates an error handling routine by specifying where normal execution is to resume.
RESUME 0 or RESUME without a line number causes the computer to return to the statement in which the error occurred. If RESUME is followed by a line number, it causes the computer to branch to the line number provided.
RESUME NEXT causes the computer to branch to the statement following the point at which the error occurred.

Example

```
10 ON ERROR GOTO 80
20 PRINT "SIMPLE DIVISION."
30 INPUT "ENTER TWO NUMBERS ";A,B
40 IF A = 0 END
50 C = A/B
60 PRINT "THE QUOTIENT IS ";C
70 GOTO 20
80 PRINT "ATTEMPT TO DIVIDE BY ZERO !"
90 PRINT "TRY AGAIN..."
100 RESUME 20

READY
>RUN

SIMPLE DIVISION.
ENTER TWO NUMBERS ? 6 , 2
THE QUOTIENT IS  3
SIMPLE DIVISION.
ENTER TWO NUMBERS ? 7 , 3
THE QUOTIENT IS  2.33333
SIMPLE DIVISION.
ENTER TWO NUMBERS ? 5 , 0
ATTEMPT TO DIVIDE BY ZERO !
TRY AGAIN...
SIMPLE DIVISION.
ENTER TWO NUMBERS ? 9 , 4
THE QUOTIENT IS  2.25
SIMPLE DIVISION.
ENTER TWO NUMBERS ? 0 , 0


READY
```

3.29 REM

REM represents remarks. This statement informs the computer that the rest of the line only consists of comments, and should be ignored. The statement also allows the user to have more comments in his program for better documentation. If REM is used in a multi-statement program line, it must be the last statement.

Example

```
10 REM  *  VARIABLE REPRESENTATIONS *
20 REM  *  A = AMOUNT               *
30 REM  *  B = NUMBER OF ITEMS      *
40 REM  *  C = UNIT COST            *
50 REM  * ------------------------- *
60 A = B * C : REM  **  AMOUNT = NO. OF ITEMS  X  UNIT COST
```

3.30 IF expression action-clause

This statement instructs the computer to test a logical or relational expression. If the expression is TRUE, control will proceed to the "action" clause immediately following the expression. If the expression is False, control will jump to the matching ELSE statement (if there is one) or down to the next program line.
In numerical terms, if the expression, has a non-zero value, it is always equivalent to a logical true.

Example

```
10 INPUT "ENTER A VALUE (MAX. 20) ";A
20 IF A > 20 GOTO 60
30 A = A * 3.1416 * 2
40 PRINT "THE CIRCUMFERENCE IS :";A
50 END
60 PRINT "NUMBER TOO BIG ! (MAX. 20)": GOTO 10


READY
>RUN

ENTER A VALUE (MAX. 20) ? 24
NUMBER TOO BIG ! (MAX. 20)
ENTER A VALUE (MAX. 20) ? 18
THE CIRCUMFERENCE IS : 113.098
```

In this example, if A is greater than 20 then a warning is printed and another input is expected. However, if A is equal to or less than 20, the computer will go to the next line and compute the value of A, without passing through the warning message and the GOTO statement.

Example

```
120 INPUT A: IF A = 10 AND A > B THEN 160

120 INPUT A: IF A = 10 AND A > B GOTO 160
```

The two statements above have the same effect.

3.31 THEN statement or line number

Initiates the "action clause" of an IF – THEN type statement. THEN is optional except when it is used to specify a branch to another line number, as in IF A > D THEN 100. THEN should also be used in IF – THEN – ELSE statements.

3.32 ELSE statement or line number

This statement must be used after the IF statement, and acts as an alternative action in case the IF test fails.

Example

```
10 IF A = 1 THEN 60 ELSE 40
```

In this example, if A = 1 then control branches to line 60, otherwise it branches to line 40. If the ELSE clause is not used and A is not equal to 1, the computer will go to the next statement instead of branching to line 40.
IF-THEN-ELSE statements may be nested, but the number of IFs and ELSEs must match with each other.

Example

```
10 INPUT "ENTER THREE NUMBERS ";X,Y,Z
20 PRINT "THE LARGEST NUMBER IS :";
30 IF X < Y OR X < Z THEN IF Y < Z THEN PRINT Z ELSE PRINT Y ELSE PRINT X
40 END
READY
>RUN
ENTER THREE NUMBERS ? 30 , 75 , 73
THE LARGEST NUMBER IS : 75
```

This program accepts three numbers and prints out the one that has the highest value.

3.33 LPRINT

Prints a file onto the printer. This command (and statement) functions similar to a PRINT statement (print on the display). If the line printer is not properly connected, the computer will enter a dead loop and will wait to print the first character. This situation can only be resolved by turning the printer on or hitting the RESET button.

```
10 FOR X = 1 TO 0 STEP -0.25
20 LPRINT "THE VALUE OF X :";X
30 NEXT X
40 END
READY
>RUN
THE VALUE OF X : 1
THE VALUE OF X : .75
THE VALUE OF X : .5
THE VALUE OF X : .25
THE VALUE OF X : 0
```

CHAPTER 4

PROCESSING ARRAYS

An array is simply an ordered list of data. Both alphabetic and numeric arrays are acceptable by the System 80. However, the data type of an array must be consistent. The concept of arrays is very important in computer programming, therefore the user should try to understand the examples in this Chapter.

Suppose John Washington is studying in a college. There is a three story building which has four classrooms on each floor and each room has 45 seats.
John is taking a history course. There are only 36 students in his class. Now, let us look through the name list of John's class.

NAME LIST

1. Mary Adams
2. Jimmy Brown
3. Henry Cox

.
.
.
.
.

36. John Washington

In order to find a specific person in the list, we just read the list from top to bottom or from bottom to top; however, the method of searching by name is not very important here.

The most important issue is how we can find a person in the list by referring his number only. In the list mentioned above, the 1st person is Mary Adams, the 2nd person is Jimmy Brown, etc. The numbers give us a systematic way to find a person.

If we use a computer to record this list, we may assign each name in the list to a unique variable, as the following.

```
10 N0$ = "MARY ADAMS"
20 N1$ = "JIMMY BROWN"
30 N2$ = "HENRY COX"
40 .
50 .
60 .
70 .
80 .
90 .
100 N5$= "TOM HUDSON"
110 .
120 .
130 .
140 NZ$ = "JOHN WASHINGTON"
```

This is a time consuming and inefficient method; besides, what happens if there are 37 students in the class?
Obviously, we need to use a variable name starting with another letter, such as M1$, etc. Another way, also the better way to handle this list is by using an array. We first define an array AR$ of 45 elements (for there are 45 seats), then assign those names to each element.

Example

```
5  CLEAR 1000 :        REM   CLEAR 1000 BYTES FOR STRING STORAGE.
10 DIM AR$(44) :       REM   ARRAY AR$ HAS 45 ELEMENTS.
20 FOR N = 0 TO 44 :   REM   LOOPS 45 TIMES
30 INPUT "ENTER THE NAME OF THE STUDENT ";AR$(N)
40 REM   ASSIGN THE NAMES TO EACH ELEMENT IN THE ARRAY.
50 NEXT N
60 END
```

This program accepts 45 names and stores them in the array AR$. After executing the program, the following should be true.

Element AR$ (0) has the value of "Mary Adams"

Element AR$ (1) has the value of "Jimmy Brown"

Element AR$ (2) has the value of "Henry Cox"

.
.
.

Element AR$ (36) has the value of "John Washington"

Provided the inputs are correct, of course!
Now, if we want to print out the entire list, we may use this program.

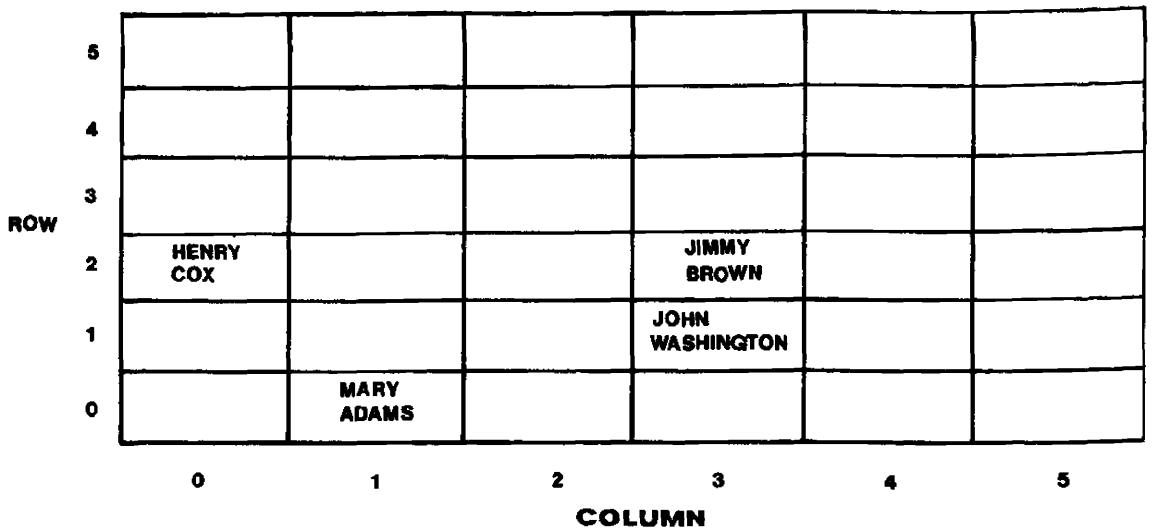
```
5  CLEAR 1000 :        REM   CLEAR 1000 BYTES FOR STRING STORAGE.
10 DIM AR$(44) :       REM   ARRAY AR$ HAS 45 ELEMENTS.
15 REM   **  INPUT ARRAY SECTION  **
20 FOR N = 0 TO 44 :  REM   LOOPS 45 TIMES
30 INPUT "ENTER THE NAME OF THE STUDENT "; AR$(N)
40 REM   ASSIGN THE NAMES TO EACH ELEMENT IN THE ARRAY.
50 NEXT N
55 REM   **  PRINT ARRAY SECTION  **
60 FOR N = 0 TO 44 : REM   LOOPS 45 TIMES.
70 PRINT AR$(N) :     REM   PRINTS THE N TH ELEMENT OF THE ARRAY.
80 NEXT N
90 END
```

Instead of the following statements.

```
10 PRINT N0$
20 PRINT N1$
30 PRINT N2$
40 .
50 .
60 .
70 .
80 PRINT N5$
90 .
100 .
110 PRINT N2$
120 .
130 .
```

By now, the user should have some feeling of how powerful arrays could be.

Suppose the teacher in John's class wants to set up a seat plan by rows and colums. Since there are 6 columns, then only 6 rows of seats are needed.



The four students we always mentioned are seated as in the plan above. Since they arc not seated according to the name list, we need another method to access the seat plan. For example, if the professor tries to see if John Washington is absent or not, he has to look through the room and find out whether the seat at row 1, column 3 is empty or not. The professor has to search for row 2 column 0 for Henry Cox as well. Actually, the computer just works the same as the teacher does. We may map this seat plan into a two dimensional array named SP$ (5, 5), the first 5 is for row, and the second 5 is for column. In case we want to call Jimmy Brown, we must reference SP$ (2, 3), that is row 2 column 3.

Now suppose we want to print the seat plan in a table form, we may use the program below:

```
10 CLEAR 1000: DIM SP$(5,5) : REM  SP$ IS A 6 X 6 ARRAY.
20 FOR R = 5 TO 0 STEP -1
30     REM   SET A LOOP TO PRINT FROM ROW 5 TO ROW 0.
40     FOR C = 0 TO 5
50         REM   SET A LOOP TO PRINT THE NAMES IN EACH COLUMN.
60         PRINT SP$(R,C), : REM   PRINT THE NAME AT ROW 'R' COLUMN 'C'.
70     NEXT C
80     PRINT : REM   CARRIAGE RETURN
90 NEXT R
100 END
```

This program prints a seat plan in a table form. It starts with the last row in the class, and ends with the first row. The program first initializes. R = 5, C = 0 then prints the value of the elements.

```
SP$(5,0); SP$(5,1); SP$(5,2); SP$(5,3); SP$(5,4); SP$(5,5)
```

At this point, the value of C becomes 5, the computer jumps out of the loop "C" and prints a blank line as on line 80 and slips to the next line. The computer passes line 70 and loops back to line 20 and then R = 4; the computer resets C = 0 on line 40 and prints the value of.

```
SP$(4,0); SP$(4,1); SP$(4,2); SP$(4,3); SP$(4,4); SP$(4,5)
```

The process repeated until R = -1, and the program stops. The final output will have values of the elements in the following order.

```
SP$(5,0); SP$(5,1); SP$(5,2); SP$(5,3); SP$(5,4); SP$(5,5)
SP$(4,0); SP$(4,1); SP$(4,2); SP$(4,3); SP$(4,4); SP$(4,5)
SP$(3,0); SP$(3,1); SP$(3,2); SP$(3,3); SP$(3,4); SP$(3,5)
SP$(2,0); SP$(2,1); SP$(2,2); SP$(2,3); SP$(2,4); SP$(2,5)
SP$(1,0); SP$(1,1); SP$(1,2); SP$(1,3); SP$(1,4); SP$(1,5)
SP$(0,0); SP$(0,1); SP$(0,2); SP$(0,3); SP$(0,4); SP$(0,5)
```

By using this two dimensional array, we can locate the exact position of any student in a class. But how can we locate another student who sits at the identical position as John Washnighton, but in the next class? Of course, we need to mention which class or which room number that the student is in. In this case, we need another dimension to describe a specific student's location. Remember, there are a total of twelve class rooms in the building. We have different ways to solve this problem. The first method is to assign a number ranged from 1 to 12 to each room. Or we may distinguish them by floor number. That is room 1 on the 1st floor, room 2 on the 1st floor, ... , room 1 on the 3rd floor, ...etc. The first method requires only one additional dimension, whereas the second method requires two additional dimensions.

Say John's classroom is the 3rd room on the second floor. By using the first method, we may locate John by referring SPS (N, R, C) where N represents the number of the room. R represents row number and C represents column number. To be more specific, John sits at SP$ (7, 1, 3) that is room number 7, row number 1, column number 3. However, by using the second method, we need to mention SP$ (F, N, R, C) where F represents floor number, N represents the room number, R represents row number, C represents column number. To locate John, we need to refer to SP$ (2, 3, 1, 3), that is the 2nd floor. room number 3, row number 1, column number 3.

The number of dimensions may increase if we try to accept and classify more students into this set. If we try to identify some other students in another building, we need another dimension to define which building. If we consider other colleges, yet we need another dimension to describe which college.

In every System 80, the number of dimensions in an array is only limited by the memory space available in the computer.

CHAPTER 5

STRING HANDLING

String operations are the essence in data processing.
It is obvious that if a computer cannot handle string operations, it is only a super powerful calculator. Based on this fact, the System 80 allows many useful string operations in addition to arithmetic operations.

In this chapter, we will discuss various string functions that are acceptable in our Extended Basic language.

5.1 String Comparison

By using a relational operator, two strings may be compared for equality or alphabetic precedence. If they are checked for equality, every character, including any leading or trailing blanks, must be identical otherwise the test fails.

Example

```
100 IF A$ = "YES" THEN 250
```

Strings are compared character by character from left to right. Actually, the ASCII code representations for the characters are compared. A character with the lower code number is considered to precede the other character. In other words, "AB" precedes "AC". When strings of different lengths are compared, the shorter string is precedent even if its characters are identical as those in the longer string. Therefore, "B" precedes "B ". The following relational operators may be used to compare strings.

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

5.2 String Operation

Basically, there is only one string operation, that is concatenation which is represented by the plus sign "+".

Example

```
10 S1$ = "THE SUN IS"
20 S2$ = " SHINING"
30 S3$ = ","
40 C$ = S1$ + S2$ + S3$ + S2$ + S3$ + S2$ + "."
50 PRINT C$
60 END
READY
>RUN
THE SUN IS SHINING, SHINING, SHINING.
```

5.3 ASC (string)

This statement returns the ASCII code (in decimal) for the first character of the specified string. The string specified must be enclosed in parentheses. A null-string will cause an error to occur.

```
100 PRINT "THE ASCII CODE FOR 'H' IS:"; ASC("H")
105 S$ = "HOME":PRINT "THE STRING IS:"; S$
110 PRINT "THE ASCII CODE FOR THE FIRST LETTER IS:";ASC(S$)
120 END
READY
>RUN
THE ASCII CODE FOR 'H' IS: 72
THE STRING IS:HOME
THE ASCII CODE FOR THE FIRST LETTER IS: 72
```

Both lines will print the same number.

A complete set of control, graphics, and ASCII codes is listed in appendix C.

5.4 CHR$ (expression)

This statement works as the inverse of the ASC function, that is to return the character of the specified ASCII, control or graphics code. The argument may be any number from 0 to 255, or any variable expression with a value within that range. The argument must be enclosed in parentheses.

```
100 PRINT CHR$(33)  : REM   PRINT A '!' SIGN
```

5.5 LEFT$ (string, n)

This statement returns the first n characters of the specified string. The arguments must be enclosed in parentheses. String may be a constant or an expression, and n may be a numeric expression.

Example

```
10 A$ = "ABCDEFG"
20 B$ = LEFT$(A$, 4)
30 PRINT B$
40 END
READY
>RUN

ABCD
```

5.6 RIGHT$ (string, n)

Returns the last n characters of a string. Both string and n must be enclosed in parentheses. String may be a string constant or variable, and n may be a numerical constant or variable. If the length of the string is less than or equal to n, the entire string is returned.

Example

```
10 A$ = "ABCDEFG"
20 B$ = RIGHT$(A$, 3)
30 PRINT B$
40 END
READY
>RUN

EFG
```

5.7 LEN (string)

Returns the length value of the specified string. The string may be a variable, expression or constant and must be enclosed in parentheses.

Example

```
10 A$ = "ABCDEFG"
20 PRINT "LENGTH OF THE STRING: "; LEN(A$)
30 END
READY
>RUN
LENGTH OF THE STRING:  7
```

5.8 MID$ (string, p, n)

Returns a substring of string starting at position p, with length n. The string, position and length must be enclosed in parentheses. String may be a constant or an expression, p and n may be numeric expressions or constants.

Example

```
10 A$ = "ABCDEFG"
20 B$ = MID$(A$,3,4)
30 PRINT "THE NEW STRING IS : ";B$
40 END

READY
>RUN

THE NEW STRING IS : CDEF
```

5.9 STR$ (expression)

Converts a constant or numeric expression into a string of characters. The expression or constant must be enclosed in parentheses.

Example

```
10 A = 34.56
20 B$ = STR$(A)
30 B$ = B$ + "%"
40 PRINT "THE RESULT IS ";B$
50 END
READY
>RUN

THE RESULT IS  34.56%
```

5.10 STRING$ (n, character or number)

Returns a string which composed of n number of the specified character.

Example

```
10 PRINT STRING$(10, "*")
20 END

READY
>RUN

**********
```

Character may be a number from 0-255; in this case, it will be treated as an ASCII, control or graphics code.

```
10 PRINT STRING$ (10, 33)
20 END


READY
>RUN

!!!!!!!!!!
```

5.11 VAL (string)

Performs the inverse of the STR$ function; that is to return the numeric value of the characters in a string argument.

Example

```
10 A$ = "56"
20 B$ = "23"
30 C = VAL (A$ + "." + B$)
40 PRINT "THE RESULTS ARE :"; C; ","; C+100
50 END


READY
>RUN
THE RESULTS ARE : 56.23 , 156.23
```

CHAPTER 6

BUILT-IN ARITHMETIC FUNCTIONS

In this chapter, we will discuss the built-in functions available in the System 80. In most cases, it is necessary to pass an argument (initial value) to the function, before a desired value (result) would be returned. The argument may be a constant, a numeric variable, or an expression. The general format could be:

result = function (argument)

Example

```
10 A = RND (3)
20 B = INT (C) / D
30 E = SQR (F * G - 4)
```

Functions discussed in this chapter:

1. ABS(X)
2. ATN(X)
3. CDBL(X)
4. CINT(X)
5. COS(X)
6. CSNG(X)
7. EXP(X)
8. FIX(X)
9. INT(X)
10. LOG(X)
11. RANDOM
12. RND(X)
13. SGN(X)
14. SIN(X)
15. SQR(X)
16. TAN(X)

6.1 ABS (X)

Returns the absolute value of the argument X.

6.2 ATN (X)

Returns the arctangent function (in radians) of the argument. To get the arctangent in degrees, multiply ATN (X) by 57.29578.

6.3 CDBL (X)

Returns a double-precision representation of the argument. The value returned contains 17 digits, however, only the digits contained in the argument will be significant.

6.4 CINT (X)

Returns the largest integer that is not greater than the argument. The argument must be within the range of – 32768 to + 32768.
For example, CINT (2.6) returns 2; CINT (-2.6) returns -3.

6.5 COS (X)

Returns the cosine function of the argument (in radians). In order to obtain the cosine of X when X is in degrees, use COS (X* .0174533)

6.6 CSNG (X)

Returns a single-precision representation of the argument. It returns a 6 significant digit number with 4/5 rounding for a double precision argument.

6.7 EXP (X)

Returns the "natural exponential" of X, that is e^X.
This is the inverse of the LOG function.

6.8 FIX (X)

Returns a truncated representation of the argument with all digits on the right of the decimal point being truncated or chopped off. For example, FIX (1.5) returns 1, FIX (-1.5) returns -1.

6.9 INT (X)

Returns an integer representation of the argument, using the largest integer that is not greater than the argument. The argument is not limited to the range -32768 to +32768. For example , INT (3.5) returns 3, INT (-3.5) returns -4.

6.10 LOG (X)

Returns the natural logarithm of the argument, that is $\log_e(X)$. To find the logarithm of a number of another base b, use the formula $\log_b(X) = \log_e(X)/\log_e(b)$.

6.11 RANDOM

This function causes the computer to generate a new set of random numbers every time when the computer is turned on and runs a program which has RND functions. No argument is needed in this function.

6.12 RND (X)

Returns a pseudo-random using the current pseudo-random number (generated internally and has not access to the user).
RND (0) returns a single-precision value between 0 and 1,
RND (X) returns an integer between 1 and X inclusive.
However, X must be positive and less than 32768.

6.13 SGN (X)

The "sign" function, that is to return -1 if X is negative, 0 is X is zero, and + 1 if X is positive.

6.14 SIN (X)

Returns the sine function of the argument (in radians).
To obtain the sine of X when X is in degrees use SIN (X* .0174533).

6.15 SQR (X)

Returns the square root of the argument.

6.16 TAN (X)

Returns the tangent function of the argument (in radians).
To obtain the tangent of X and X is in degree, use TAN (X* .0174533).

CHAPTER 7

GRAPHICS FEATURES

There are only four graphics functions available in the System 80. However, they are powerful enough to allow the user to create any graphic patterns on the display with or without the help of our Extended BASIC language.

For the display map, please refer to appendix E.

7.1 SET (x, y)

This function turns on the graphics block on the display at the location specified by the coordinates x and y. The display is divided up into a 128 (horizontal) by 48 (vertical) grid. The x – coordinates are ranged from 0 to 127, organized from left to right. The y – coordinates are ranged from 0 to 47, organized from top to bottom. Therefore, point (0, 0) is located at the extreme top left corner of the display; whereas point (127, 47) is located at the extreme bottom right corner of the display. The arguments x and y may be numeric constants, variables or expressions. Since the SET (x, y) function uses only the integer portion of x and y, neither argument need be an integer.

7.2 RESET (x, y)

This function turns off a graphics block on the display at the location specified by the coordinates x and y. This function has the same limits and parameters as SET (x, y).

7.3 CLS

This function clears the entire display by turning off all the graphics blocks. It also moves the cursor to the upper left corner. This function allows the user to present an outstanding display on the screen, without any symbol previously displayed.

7.4 POINT (x, y)

This function examines the specified graphics block to see whether it is ON or OFF If the block is ON (has been SET), then POINT returns a binary True (-1). If the block is OFF, POINT returns a binary False (0).

Example

```
A = POINT (3, 40)
```

If point (3, 40) has been set, then A has the value of -1. Otherwise A has the value of 0.

CHAPTER 8

SPECIAL FEATURES

8.1 INP (port-number)

Input a 8-bit value from the specified port. The System 80 is capable of handling 256 ports, numbered from 0 to 255. Usually this function is used only when the expansion box is installed.

Example

```
10 A = INP (124)
```

This will input an 8-bit value from port 124 and assign it to variable A.

8.2 OUT port-number, value

Output an 8-bit value to the specified port. This statement requires two arguments: port-number and the value. The System 80 is capable of handling 256 ports, numbered from 0 to 255.

Example

```
30 OUT 14, 240
```

Output the value 240 to port 14. Both arguments are limited to single byte values, that is 0-255.

8.3 PEEK (address)

This function returns the 8-bit value stored at the specified decimal address in the computer's memory, and displays the value in decimal form. The value will be between 0-255.
Example

```
20 B = PEEK (30000)
```

Returns the value stored at location 30000 and assign that value to the variable B.

8.4 POKE address value

This statement sends a 8-bit value to the specified (decimal) memory address location. It requires two arguments: address and value. The value must be between 0-255.
Example

```
10 A = 250
20 POKE 19000, A      : REM   SEND VALUE OF A TO ADDRESS 19000.
30 B = PEEK (19000)   : REM   RETURNS VALUE AT ADDRESS 19000 TO B.
40 PRINT "THE RESULT IS:";B
50 END
READY
>RUN

THE RESULT IS: 250
```

8.5 MEM

Returns the number of unused and unprotected bytes in memory.

Example

```
200 IF MEM < 100 THEN 700
```

When used as a command, it must be accompanied with the PRINT command. That is PRINT MEM, to find out the amount of memory not being used to store program, variables, strings, arrays, etc.

APPENDIX A

System 80 Reserved Words*

ABS
AND
ASC
ATN
CDBL
CHR$
CINT
CLEAR
CLOSE
CLS
CONT
COS
DATA
DEFDBL
DEFFN
DEFINT
DEFSNG
DEFUSR
DEFSTR
DELETE
DIM
EDIT
ELSE
END
ERL
ERR
ERROR
EXP
FIX
FOR
FRE
GET
GOSUB
GOTO
IF
INKEY$
INP
INPUT
INSTR
INT
KILL
LEFT$
LET
LSET
LEN
LINE
LIST
LOAD
MEM
MID$
NAME
NEW
NEXT
NOT
ON
OUT
PEEK
POINT
POKE
POS
PRINT
PUT
RANDOM
READ
REM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
SET
SGN
SIN
SQR
STEP
STOP
STRING$
STR$
TAB
TAN
THEN
TROFF
TRON
USING
USR
VAL
VARPTR

*None of these words can be used inside a variable name.

APPENDIX B

ERROR CODES

CODE	ABBREVIATION	ERROR
1	NF	NEXT without FOR
2	SN	Syntax error
3	RG	Return without GOSUB
4	OD	Out of data
5	FC	Illegal function call
6	OV	Overflow
7	OM	Out of memory
8	UL	Undefined line
9	BS	Subscript out of range
10	DD	Redimensioned array
11	/0	Division by zero
12	ID	Illegal direct
13	TM	Type mismatch
14	OS	Out of string space
15	LS	String too long
16	ST	String formula too complex
17	CN	Can't continue
18	NR	NO RESUME
19	RW	RESUME without error
20	UE	Unprintable error
21	MO	Missing operand
22	FD	Bad file data

Explanation of Error Messages

NF NEXT without FOR: NEXT is used without a matching FOR statement. This error may also occur if NEXT variable statements are reversed in a nested loop.

SN Syntax Error: This is usually the result of incorrect punctuation, open parenthesis, an illegal character or a mis-spelled command.

RG RETURN without GOSUB: A RETURN statement was encountered before a matching GOSUB was executed.

OD Out of Data. A READ or INPUT # statement was executed with insufficient data available. DATA statement may have been left out or all data may have been read from tape of DATA.

FC Illegal Function Call: An attempt was made to execute an operation using an illegal parameter. Examples: square root of a negative argument, negative matrix dimension, negative or zero LOG arguments, etc. Or USR call without first POKEing the entry point.

OV Overflow: A value input or derived is too large or small for the computer to handle.

OM Out of Memory: All available memory has been used or reserved. This may occur with very large matrix dimensions, nested branches such as GOTO, GOSUB, and FOR-NEXT Loops.

UL Undefined Line: An attempt was made to refer or branch to a non-existent line.

BS Subscript out of Range: An attempt was made to assign a matrix element with a subscript beyond the DIMensioned range.

DD Redimensioned Array: An attempt was made to DIMension a matrix which had previously been dimensioned by DIM or by default statements. It is a good idea to put all dimension statements at the beginning of a program.

/O Division by Zero: An attempt was made to use a value of zero in the denominator.

ID Illegal Direct: The use of INPUT as a direct command.

TM Type Mismatch: An attempt was made to assign a non-string variable to a string or vice-versa.

OS out of String Space: The amount of string space allocated was exceeded.

LS String Too Long: A string variable was assigned a string value which exceeded 255 characters in length.

ST String Formula Too Complex: A string operation was too complex to handle. Break up the operation into shorter steps.

CN Can't Continue: A CONT was issued at a point where no continuable program exists. e.g. after program was ENDed or EDITed.

NR NO RESUME: End of program reached in error-trapping mode.

RW RESUME without ERROR: A RESUME was encountered before ON ERROR GOTO was executed.

UE Unprintable Error: An attempt was made to generate an error using an ERROR statement with an invalid code.

MO Missing Operand: An operation was attempted without providing one of the required operands.

FD Bad File Date: Data input from an external source (i.e. tape) was not correct or was in improper sequence, etc.

APPENDIX C

Control Codes: 1-31

Code	Function
8	Backspaces and erases current character
9	None
10-13	Carriage returns
14	Turns on cursor
15	Turns off cursor
16-22	None
23	Converts to 32 character mode
24	Backspace ← Cursor
25	Advance → Cursor
26	Downward ↓ linefeed
27	Upward ↑ linefeed
28	Home, return cursor to display position (0,0)
29	Move cursor to beginning of line
30	Erases to the end of the line
31	Clear to the end of the frame

ASCII Character Codes 32-128

Code	Character	Code	Character
32	space	65	A
33	!	66	B
34	"	67	C
35	#	68	D
36	$	69	E
37	%	70	F
38	&	71	G
39	'	72	H
40	(	73	I
41	)	74	J
42	*	75	K
43	+	76	L
44	,	77	M
45	-	78	N
46	.	79	O
47	/	80	P
48	0	81	Q
49	1	82	R
50	2	83	S
51	3	84	T
52	4	85	U
53	5	86	V
54	6	87	W
55	7	88	X
56	8	89	Y
57	9	90	Z
58	:	91	[
59	;	92	\
60	<	93	]
61	=	94	^
62	>	95	_
63	?	96-127	Lower case for codes 64-95
64	@	128	Space

APPENDIX D

Program Limits and Memory Overhead

Ranges

Integers	32768 + 32767 inclusive
Single Precision	−1.701411E + 38 to + 1.701411E + 38 inclusive
Double Precision	−1.701411834544556E + 38 to + 1.701411834544556E + 38 inclusive

String Range: Up to 255 characters

Line Numbers Allowed: 0 to 65529 inclusive

Program Line Length: Up to 255 characters

Memory Overhead
Program lines require 5 bytes minimum, as follows:
- Line Number – 2 bytes
- Line Pointer – 2 bytes
- Carriage Return – 1 byte

In addition, each reserved word, operator, variable name, special character and constant character requires one byte.

Dynamic (RUN-time) Memory Allocation

Integer variables: 5 bytes each
(2 for value, 3 for variable name)

Single-precision variables: 7 bytes each
(4 for value, 3 for variable name)

Double-precision variable:
(8 for value, 3 for variable name)

String variables: 6 bytes minimum
(3 for variable name, 3 for stack and variable pointers, 1 for each character)

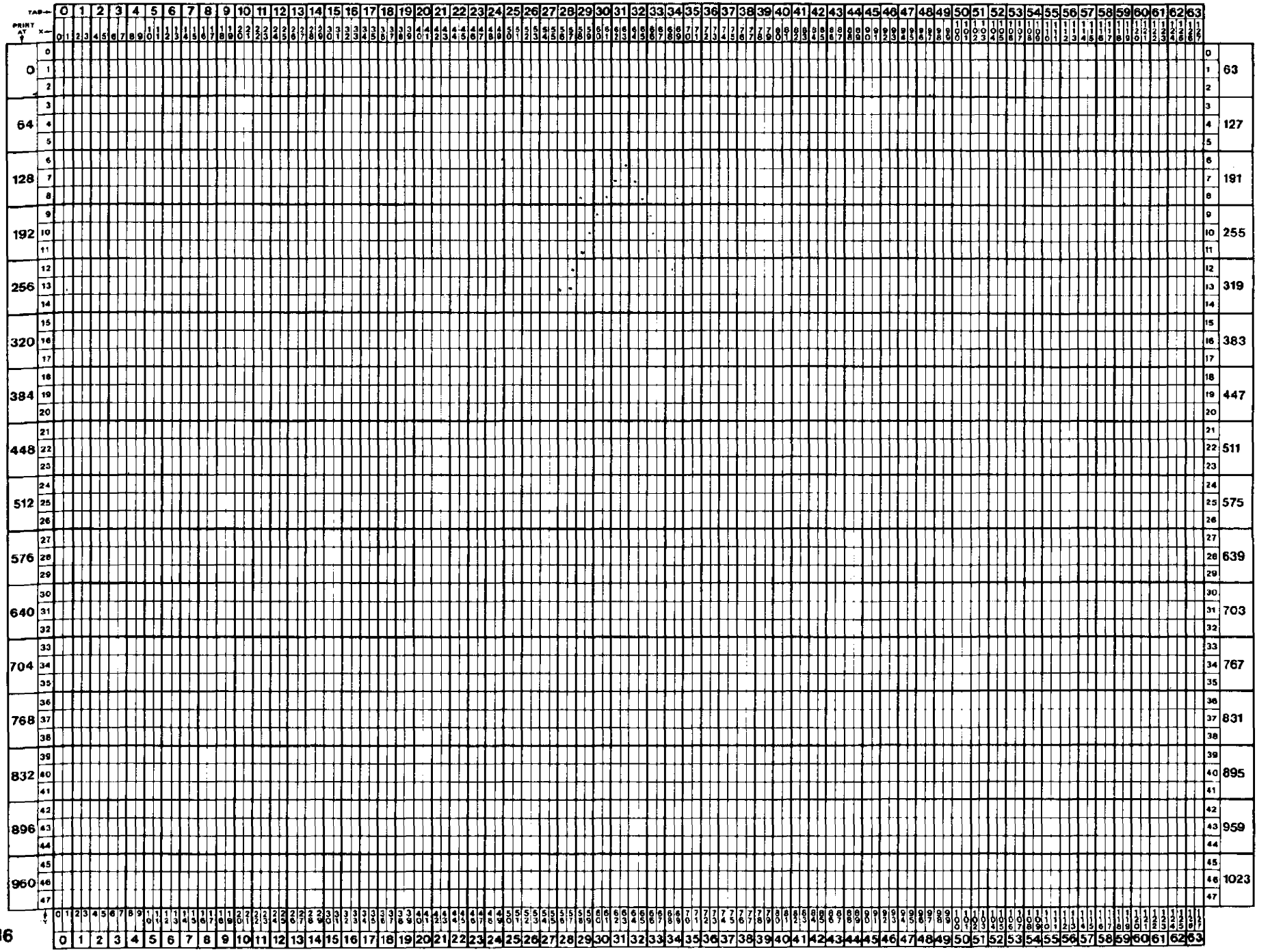
Array variables: 12 bytes minimum
(3 for variable name, 2 for size, 1 for number of dimensions,
2 for each dimension, and 2,3,4, or 8 [depending on array type]
for each element in the array)

Each active FOR-NEXT loop requires 16 bytes.

Each active (non-returned) GOSUB requires 6 bytes.

Each level of parentheses requires 4 bytes plus 12 bytes for each temporary value.

APPENDIX E VIDEO DISPLAY MAP



Four more BASIC commands should be included in the instruction set. They are (1) INKEY$, (2) POS, (3) USR and (4) VARPTR.

ADDENDUM

(1) INKEY$

Returns a one-character string determined by an instantaneous input from the keyboard. If no key is pressed during the execution of this statement, a null string is returned.

Characters typed to an INKEY$ are not automatically displayed on the screen.

Example:

```
10 REM * ENTER A PASSWORD WITHOUT
20 REM * DISPLAYING IT ON THE SCREEN
30 CLS
40 PRINT "INPUT A PASSWORD "
50 A$=INKEY$: IF A$="O" THEN 60 ELSE 50
60 B$=INKEY$: IF B$="K" THEN 70 ELSE 60
70 PRINT "WELCOME !!! "
```

(2) POS (dummy argument)

The computer returns a number from 0 to 63 indicating the current cursor position on the display. Usually, 0 is used for the dummy argument.

Example:

```
100 A=POS(0)
```

(3) USR (argument)

Calls a machine language subroutine and passes the argument to the subroutine. Such a subroutine could be loaded from tape or created by POKEing Z80 machine code into the memory. Users who are not familiar with machine language programming are not recommended to use this command.

The subroutine entry address should be POKEd into location 16526 – 16527. The least significant byte should be in location 16526.

To pass the argument to the subroutine, the subroutine should immediately execute a CALL OA7FH (call 2687 dec.). The argument will then be placed in registers HL.

To return to your BASIC program without passing any value back, a RET instruction should be executed.

To return a value, load the value into the HL register pair as a two-byte signed integer and execute a JP OA9AH instruction. (OA9HA = 2714 Decimal)

USR routine reserves 8 stack levels for the users' subroutine.

Example:

```
10 INPUT I% : REM * INPUT ARGUMENT *
15 REM * PREPARE ENTRY ADDRESS *
20 POKE 16526,0 :POKE 16527,120
30 A=USR(I%) : REM * RETURN ARGUMENT A *
```

The subroutine should place on top of the memory map. To protect that region of memory, the user should input the highest memory location available for his BASIC program storage when the machine asks READY? at power up.

(4) VARPTR (variable name)

An address – value of the variable name will be returned.

If K is the returned address, the variables will be stored in the following structures :-

(i) 2 – byte integer
K – LSB
K + 1 – MSB

(ii) single precision variable
K – LSB
K + 1 – Next MSB
K + 2 – MSB
K + 3 – Exponent value

(iii) double precision value
K – LSB
K + 1 – Next MSB
•
•
•
K + 6 – MSB
K + 7 – Exponent value

(iv) string variable
K – length of string
K + 1 – LSB of string starting address.
K + 2 – MSB of string starting address.

ACTIVE COMMANDS

AUTO	13
CLEAR	14
CLOAD	15
CLOAD?	15
CONT	15
CSAVE	16
DELETE	16
EDIT	16
LIST	17
LLIST	19
NEW	17
RUN	17
SYSTEM	18
TROFF	18
TRON	18

PROGRAMMING COMMANDS

	Page		Page
CLEAR	45	LET	47
DATA	39	LPRINT	60
DEFDBL	44	ON n GOSUB	52
DEFINT	43	ON n GOTO	50
DEFSNG	44	ON ERROR GOTO	50
DEFSTR	45	PRINT	28
DIM	45	PRINT @	30
ERROR	55	PRINT TAB	31
END	47	PRINT USING	31
FOR NEXT	52	PRINT#	41
GOSUB	50	READ	39
GOTO	49	RESTORE	40
IF THEN ELSE	59	RETURN	50
INKEY$	*	RESUME	57
INPUT	36	REM	58
INPUT#	47	STOP	48

EDITING COMMANDS

NEWLINE	–	record all changes
SPACEBAR	–	move cursor one space to the right
BACKSPACE	–	move cursor back to the left
SHIFT-ESC	–	escape from Insert command
H	–	hack and insert
I	–	insert
X	–	insert at end of line
L	–	list line
A	–	cancel all editing changes
E	–	save all editing changes
Q	–	back to Active Command level with no change
D	–	delete
C	–	change
S	–	search
K	–	delete specified characters

STRING FUNCTIONS

	Page
ASC	68
CHR$	69
LEFT$	69
LEN	70
MID$	70
RIGHT$	69
STR$	71
STRING$	71
VAL	72

ARITHEMETIC FUNCTIONS

Page 73

ABS	INT
ATN	LOG
CDBL	RANDOM
CINT	RND
COS	SGN
CSNG	SIN
EXP	SQR
FIX	TAN

GRAPHIC FUNCTIONS

Page 76

CLS
POINT
RESET
SET

SPECIAL FUNCTIONS

Page 77

INP
OUT
PEEK
POKE
POS*
MEM
USR*
VARPTR*

*explained in ADDENDUM

Dick Smith Electronics Pty Ltd.,
Cnr Lane Cove Rd & Waterloo
Road, North Ryde NSW 2113
Australia

COPYRIGHT (C) BY EACA ,1980,
ALL RIGHTS RESERVED.