BASIC Reference Manual

for SVI·318/328
Personal Computer System

SVI™
SPECTRAVIDEO

SPECTRAVIDEO BASIC REFERENCE MANUAL

BASIC by MICROSOFT

Published by
SPECTRAVIDEO INTERNATIONAL LIMITED

Second Edition
First Printing 1984
Printed in Hong Kong

Printed in Hong Kong

Price: US$19.95
SVI-318/328-BRM

INTRODUCTION

This manual assumes the reader has some knowledge of the programming language, BASIC. It is recommended that the reader has BASIC up and running as this manual is read through, to try out those commands and programs given as examples.

Chapter 1 explains the various versions of BASIC.

Chapter 2 describes the essential knowledge to start running BASIC: means of starting different versions of BASIC, modes of operation and functions of all keys on keyboard.

Chapter 3 is the threshold of programming. It starts with definitions of terms and error messages in programming, arithmetic and relational operation, to various input/output.

Chapter 4 is the heart of this manual, it explains and provides example for every individual command, statement or function.

The appendix hold those handy information need to be referred to occasionally. In particular, Appendix B on Disk BASIC is essential to programming.

To familiarize with SV BASIC and some fundamental techniques, read through the first three chapters, before going to Chapter 4 and Appendix.

FORMAT NOTATION

Whenever the format of a statement or command is given, the following rules apply:

1. Items in capital must be input as shown.
2. Items in lower case letters enclosed in bracket (< >) are to be supplied by the user.
3. Items in square brackets ([ ]) are optional.
4. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hypens, equal signs) must be included where shown.
5. Items followed by an ellipse (...) may be repeated any number of times (up to the length of line).
6. "string" means a string expression.
7. "exp" means a numeric expression, either constant or variable.
8. "n" means an integer.
9. "x, y" denotes X, Y co-ordinate of the screen.
10. "∧" means $\boxed{\text{CTRL}}$ key.

Whenever an example to be tried, the following rules apply:

1. Statement or command in black bold letter are typed in through the computer's keyboard.

2. Characters or "Ok" prompt in blue are computer's response to your command.

3. Special key enclosed in a box should be pressed after input from keyboard.

TABLE OF CONTENT

CHAPTER 1

VERSION OF BASIC

1.1 Cassette BASIC

The cassette version of BASIC is built into your SVI computer in 32K bytes of read-only memory. You can use cassette BASIC on a SVI computer with any amount of random access memory. The amount of storage you can use for programs and data depends on how much random-access memory you have in your computer.

The only storage device you can use to save information in Cassette BASIC is a cassette tape recorder.

Both Cassette and Disk versions of BASIC possess the following features:

* An extended character set of 102 different characters which can be displayed. In addition to the conventional alphabets, numbers, punctuations, you will also find symbols which are commonly used in scientific and mathematical applications.

* Graphics capability. With the installed Video Display Processor TMS9918/9929 you can draw points, lines, ellipses and even entire pictures. There are 52 graphic

symbols initiated by pressing either LEFT GRPH or RIGHT GRPH with the 26 alphebet keys simultaneously. 32 sprites - user programmable pictorial shapes - are available. Screens in high or low resolution are points addressable.

* Audio capability. With the installed Sound Programmable Generator AY8910, some sound or piece of music can be produced.

* Special input/output devices. BASIC supports joysticks, paddles and graphic tablet which make your program more interesting and funny.

1.2 Disk BASIC

This version of BASIC comes as a program on the Disk BASIC diskette. You must load Disk BASIC into memory before you can use it. It requires 8008 bytes to boot the diskette. The amount of storage you can use for programs and data is displayed on the screen when you start BASIC.

Features of Disk BASIC are:

* Input/output to diskette in addition to cassette.

* Other features for Cassette BASIC.

CHAPTER 2

GETTING STARTED

2.1 To Start Different Versions of BASIC

2.1.1 To Start Cassette BASIC

Hook up your TV set or monitor to the console. Refer to the user's manual for details. If you wish to get program from tape or save a program onto cassette, connect your data cassette to the computer. This is simple. Just insert the connector located on the tail end of the cable attached to the recorder into the slot on the back of the computer.

If you have disk drive(s) connected to the computer, ensure no diskette is placed in drive 1 or else the drive door is left open. You will find the statement "SV extended BASIC version 1.0 Copyright 1983 (c) by Microsoft Corp." appear on the top screen. A list of first five function keys are displayed.

2.1.2 To Start Disk BASIC

Hook up your expander, floppy disk controller (if this is not built into the expander) and disk drive to the computer. Do not omit the television set or monitor of course. Refer to user's manual for expander or disk drive for details. Turn on the television set or monitor and expander. A few clacking noises will come from the disk drive. Before powering on the computer, insert the Disk BASIC Diskette in drive 1, with its label facing up and towards the slot. Move the lock to the vertical position. A red indicator labelled "IN USE" will light up. The statement "SV extended BASIC version 1.1 Copyright 1983 (c) by Microsoft Corp." followed by "Disk version 1.0 by Microsoft Corp." are displayed.

If you fail to load Disk BASIC, check whether enough memory is available. For SVI-318 user, an additional RAM (Random Access Memory) cartridge should be installed. 8008 bytes is required to boot up the diskette.

Whenever failure is encountered, reboot the system, ie. power off the console and then power on.

2.2 MODES OF OPERATION

Once BASIC is started, the "Ok" prompt is displayed. This signals that you may enter your command or program. Such status is known as command level. Now you may communicate with BASIC in either two modes: direct or indirect mode.

2.2.1 Direct Mode

In this mode, your command will not preceded by a line number. Your command is executed immediately. The followings may be performed in direct mode: arithmetic calculation, logical operation, variable assignment (stored for later usage) and simple command. However such instructions are lost right after execution. For example:

```
A = 34 [ENTER]
Ok
PRINT A [ENTER]
34
Ok
```

Note: Pressing $\boxed{\text{ENTER}}$ means you have finished your input and expect a response from the computer.

2.2.2 Indirect Mode

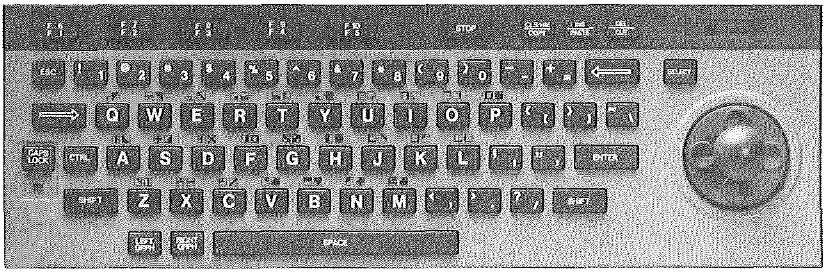
A program is entered in indirect mode. Each statement is preceded by a line number. The line is stored as part of the program in memory. The program will only be executed by entering RUN command. For Example:

```
10 A = 34
20 PRINT A
RUN [ENTER]
34
Ok
```

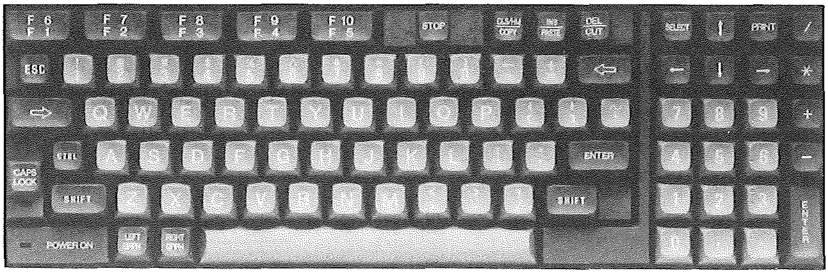
2.3 KEYBOARD

Programming is generally done by sending instructions to the computer through the keyboard. Both input instructions and the computer's responses are visible on the screen, which is connected to the console. The computer's keyboard resembles that of a typewriter. However it contains additional keys which are necessary to communicate effectively with the computer.

The 318 Keyboard



The 328 Keyboard



Basically the keyboard can be divided into five general areas:

* The function key labelled F1 through F10, are on the upper row of the keyboard.

* The "typewriter" area lies in the central part. Here you find the standard keys usually appeared on a typewriter keyboard.

* The program control keys.

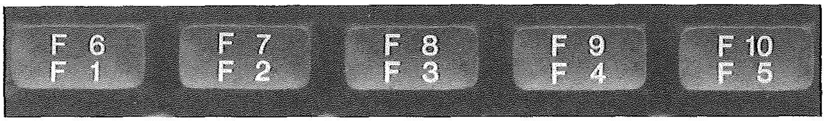
* The editing keys are located on the periphery of the keyboard.

* Special Keys.

* Built-in joystick for SVI-318 or numeric keypad for SVI-328 on the right side of console.

2.3.1 Function Key

Look at the top row of keys:



These keys are called function keys and each one is marked with the letter "F". They are a labor-saving device because they allow you to instruct the computer to perform a frequently used function by pressing only one key instead of having to type many keys.

The function keys can be used:

* As "soft keys". You can set each key to automatically type any sequence of characters or any frequently-used commands. You may use KEY statement to re-assign these keys.

* As program interrupts through use of ON KEY statement.

Here is a list of each key, the function it performs and a brief description of the function. Refer to chapter 4 for details. Function keys F1 through F5 are operated by pressing the appropriate key. Function keys F6 through F10 are operated by pressing the $\boxed{\text{SHIFT}}$ key and holding it down while simultaneously pressing the appropriate key.

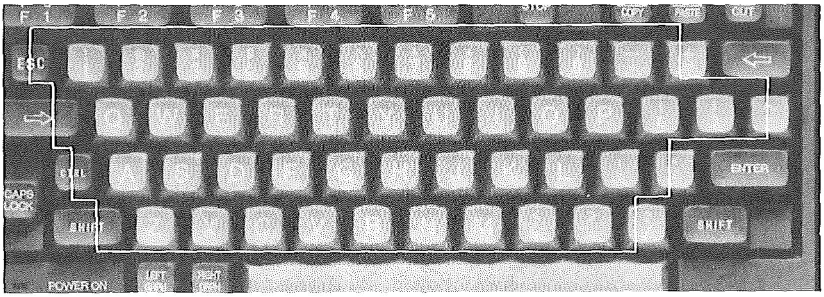
For Cassette BASIC

Key	PRE-DEFINED FUNCTION	DESCRIPTION
F1	files $\boxed{\text{ENTER}}$	Display the names of files residing on a diskette.
F2	load "1:	Load a program from diskette in disk drive 1. The filename should be specified right after colon.
F3	save "1:	Save a program file on diskette which is inserted in disk drive 1. Filename should be supplied right after colon.
F5	run $\boxed{\text{ENTER}}$	Execute the program currently resided in memory.
F6	color 15,4,5 $\boxed{\text{ENTER}}$	Print white characters on a blue background with a blue border. These colors are the default as your computer is turned on.
F7	cload	Load program from a cassette recorder.
F8	cont $\boxed{\text{ENTER}}$	Continue program execution after the last executed line.
F9	list. $\boxed{\text{ENTER}}$	Display the last line you are working on.
F10	$\boxed{\text{CLS}}$ run $\boxed{\text{ENTER}}$	Similar to F5, except that the screen is cleared before program execution.

For Disk BASIC

KEY	PRE-DEFINED FUNCTION	DESCRIPTION
F1	color	Change the text, background and border colors on your TV set/monitor.
F2	auto $\boxed{\text{ENTER}}$	To generate program line numbers automatically.
F3	goto	Execute the program currently resided in memory from any place (line number).
F4	list	Print all or part of your immediately preceding program statements on screen.
F5	run $\boxed{\text{ENTER}}$	Execute the program currently resided in memory.
F6	color 15,4,5 $\boxed{\text{ENTER}}$	Print white characters on a blue background with a blue border. These colors are the default when you turn the computer on.
F7	cload"	Load program from a cassette recorder. Supply filename to be retrieved right after the quotation mark.
F8	cont $\boxed{\text{ENTER}}$	Continue program execution after the last executed line.
F9	list. $\boxed{\text{ENTER}}$	Display the last line you are working on.
F10	$\boxed{\text{CLS}}$ run $\boxed{\text{ENTER}}$	Similar to F5, except that the screen is cleared before program execution.

2.3.2 Typewriter Key



This portion resembles a standard typewriter keyboard. It consists of the followings:

* Uppercase and lowercase alphabets. The default is lowercase. On pressing CAPS LOCK with its red light on, any alphabet printout on screen is capital type. Release the lock by pressing it a second time. The light turns off and now lowercase alphabet printout is available. Also the SHIFT key when pressed simultaneously with an alphabet key, will generate capital letter.

* Numerals from 0 to 9. For SVI-328 computers, an additional set of numerals is found in the numeric keypad.

* Symobls: punctuction, arithmetic and logical operators.

CHARACTER	ACTION
!	Exclamation point
@	At sign
#	Number sign
$	Dollar sign
%	Percent
	Up arrow or exponentiation symbol
&	Ampersand
*	Asterisk or multiplication symbol
(	Left parenthesis
)	Right parenthesis
_	Underscore
-	Hyphen or minus sign
=	Equal sign or assignment symbol
[	Left square bracket
]	Right square bracket
	Left bracket
	Right bracket
	Back slash
:	Colon
;	Semi-colon
'	Single quotation mark or apostrophe
,	Comma
.	Period, decimal point or full-stop
	Less than
	Larger than
/	Slash or division symbol
?	Question mark

* Spacebar serves two purposes:

  (i) Leave a space.
  (ii) Act as a special keyboard input in programming. Refer to STRIG(0) function for details.

* $\boxed{\text{SHIFT}}$ By holding down this key while pressing the desired key, the capital alphabets and uppershift symbols can be generated.

* $\boxed{\text{CAPS LOCK}}$ Similar to a shift lock key, but provide only capital alphabets. It cannot generate upper shift characters on the numeric or symbolic keys. It toggles between uppercase/lowercase alphabets. This key serves also as a diagnostic check indicator. As computer is switched on, an automatic system functional check is undergone. This is indicated by the temporary lighting up of this key. If the system is at fault, it will continue to illuminate. In such case, turn off all power and check all connections before powering on once more.

2.3.3 PROGRAM CONTROL KEY

The following keys are used to control the operation of computer programs.

$\boxed{\text{STOP}}$ Press this key to pause the computer after you have instructed it to execute or to perform a function, which makes it

	begin working on your program. Press it the second time to instruct the computer to resume working on your program or a function.
ENTER	Press this key at the end of each instruction you type. By pressing this key you are telling the computer to enter the instruction you just typed into its work space. The ENTER key is not used to advance the cursor to the next screen line and therefore should not be confused with the return key on a typewriter. In the event that an instruction contains more characters than can fit on a single screen line, the computer will automatically advance the cursor to the next screen line.
CTRL STOP	This tells the computer to stop what's doing and turn control back over to you, so that further instructions can be issued.

2.3.4 EDITING KEY

These keys together with the four cursor keys are used in screen editing.

CLS/HM COPY	Pressing this key will clear the screen and move the cursor to the upper lefthand corner of the screen. When pressed together with the SHIFT key, it will moved the cursor to the upper lefthand position (Home) but will not clear the screen.

INS/PASTE

This key is used when you wish to insert characters PASTE within a line. Just move the cursor to the location where you wish to insert, then press this key and the text you type will be inserted.

DEL/CUT

Press this key to delete the character under the CUT cursor.

E S C

This key is often used in software application programs. Its usual function is to interrupt the operation of a program or to continue operation following an interrupt.

This key also is not used in BASIC. It is used in a word processor or similar application program to space forward 5 spaces to begin a paragraph.

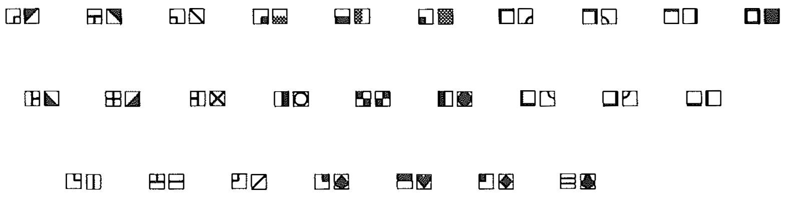
This key also is not used in BASIC. It backs up the cursor one space and deletes the character immediately to the left of the cursor prior to the key press.

2.3.5 SPECIAL KEY

LEFT GRPH

RIGHT GRPH

The LEFT GRAPH and the RIGHT GRAPH keys are used to select the graphic symbols that correspond to the keys which are displayed on the following chart. If you press the LEFT GRPH key and hold it dwon while simultaneously pressing one of the alphabet keys, the graphhic symbol above and to the left of the corresponding key on the following chart will be displayed. The corresponding symbol on the right side of the alphabet key can be displayed by pressing the RIGHT GRPH key and the corresponding key.



SELECT

PRINT

The SELECT and PRINT keys are also included on this keypad to allow the advantage of using these functions that are often available in word processing and data entry software packages. These keys have no function in BASIC programming and are only accessed from programs such as those mentioned above.

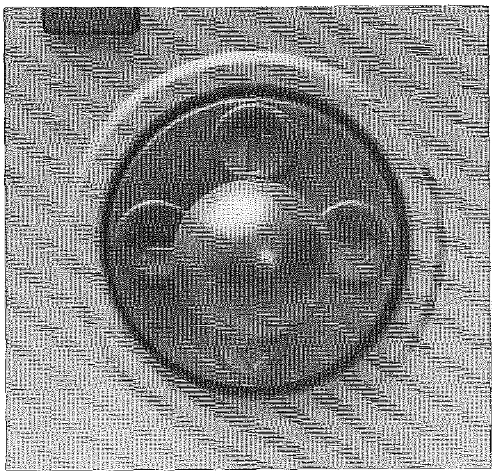
2.3.6 Arrow Key

The arrow keys (up, down, left & right) control the movement of the cursor on the display screen. By pressing a combination of the up and left arrow keys, you will cause the cursor to move towards the upper left corner of the display screen. Other combinations will work in the same fashion giving you 8 directions of cursor movement using these keys.

Cursor left (←) If the cursor advances beyond the left edge of the screen, the cursor will move to the right side of the screen on the preceding line.

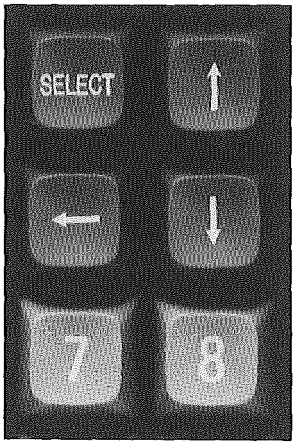
Cursor right (→) If the cursor advances beyond the right edge of the screen, the cursor will move to the left side of the screen on the next line down.

2.3.7 JOYSTICK CURSOR CONTROL PAD



The special built-in Joystick/Cursor Control pad feature is unique to the SVI-318 computer. It manipulates the cursor movement on the screen.

NUMERIC KEYPAD



The numeric keys (1-9) are the same as the keys on the top of the regular keyboard. These are used when performing rapid entry of numeric data. This keypad also contains the mathematical functions keys (+,-,*,/) which can be used to enter formulae and to perform quick calculations.

CHAPTER 3

GENERAL INFORMATION ABOUT BASIC PROGRAMMING

3.1 INTRODUCTION

To control a computer, one must give his instructions in a language that the computer understands. The SVI computers understand a language called Microsoft BASIC. BASIC stands for "Beginner's All Purpose Symbolic Instruction code". It is actually a set of English words with which you can instruct the computer to perform certain functions.

Microsoft BASIC is an extended version to the Microsoft standard BASIC version 5.3, which includes supports to graphics, music and various peripherals attached to home and personal computer. Generally, BASIC is designed to follow the GW BASIC which is a standard BASIC in the 16-bit machine world. However, great effort has been put to make the system as flexible and expandable as possible.

Also Microsoft BASIC is featured with up to 14 digits accuracy, double precision arithmetic function. This means arithmetic operations no longer generate strange round errors that confuse novice users. Every transcendental functions are also calculated with this accuracy.

3.2 WHAT IS PROGRAMMING?

Programming is the art of writing the instructions and information for the computer to read and execute. Programs differ from one another in their sets of instructions and information. Programs written in one computer language will not contain instructions and structure that a program in another computer language possesses.

There are two different ways to input a program into the computer: in program/indirect mode or immediate/direct mode.

3.3 LINE FORMAT

BASIC program lines have the following format:

〈 nnnnn 〉 BASIC statement [:BASIC statement ......][' comment] $\boxed{\text{ENTER}}$

A program line always begins with a line number nnnnn ranging from 0 to 65529. Only integer will be accepted. Line numbers indicate the order in which the program lines are stored in memory. Also they are referenced in branching and editing.

A line may contain a maximum of 255 characters. More than one BASIC statement may be placed on a line, each being separated from the last by a colon.

Comments may be added to the end of a line using the apostrophe (') to separate the comment from the rest of the line.

A program line should end by pressing $\boxed{\text{ENTER}}$.

3.4 CHARACTER SET

The character set consists of alphabets, numerals, special characters and graphic characters.

There are both upper and lower case letters for each alphabet.

Also there are ten digits, from 0 to 9.

In addition, the special characters are shown as the following table. (See page 22)

CHARACTER	ACTION
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number sign
$	Dollar sign
!	Exclamation point
[	Left square bracket
]	Right square bracket
{	Left bracket
}	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Integer division symbol
@	At sign
_	Underscore
⇦	Delete last character typed
ESC	Escape
⇨	Move print position to next tab stop. Tab stops are set every eight columns.
ENTER	Terminate input of a line.

3.5 RESERVED WORD

BASIC statements and function names are reserved. That is, the key words cannot be used in variable names. If you attempt to use any of the words listed below as the name of the variable, an error is indicated by the computer.

ABS	DEFFN
AND	DEFINT
APPEND	DEFSNG
ASC	DEFSTR
ATN	DEFUSR
ATTR$	DELETE
AUTO	DIM
BASE	DRAW
BEEP	DSKI$
BIN$	DSKO$
BLOAD	ELSE
BSAVE	END
CALL	EOF
CDBL	EQV
CHR$	ERASE
CINT	ERL
CIRCLE	ERR
CLEAR	ERROR
CLICK	EXP
CLOAD	FIELD
CLOAD?	FILES
CLOSE	FIX
CLS	FOR
COLOR	FPOS
CONT	FRE
COS	GET
CSAVE	GOSUB
CSNG	GOTO
CSRLIN	HEX$
CVI	IF
CVS	INKEY$
DATA	INP
DEFDBL	INPUT

INPUT#
INPUT$
INSTR
INT
INTERVAL
INTERVAL OFF
INTERVAL ON
INTERVAL STOP
IPL
KEY
KEY LIST
KEY ON
KEY STOP
KEY (n) OFF
KEY (n) ON
KEY (n) STOP
KILL
LEFT$
LEN
LET
LFILES
LINE
LINE INPUT#
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LPRINT USING
LSET
MAXIFILES
MERGE
MID$
MKI$
MKD$
MKS$
MOD
MOTOR ON
MOTOR OFF

NAME
NEW
NEXT
NOT
OCT$
ON ERROR GOTO
ON GOSUB
ON GOTO
ON INTERVAL GOSUB
ON KEY GOSUB
ON SPRITE GOSUB
ON STOP GOSUB
ON STRIG GOSUB
OPEN
OR
OUT
OUTPUT
PAD
PAINT
PEEK
PLAY
POINT
POKE
POS
PRESET
PRINT
PRINT USING
PRINT#
PRINT# USING
PSET
PUT SPRITE
READ
REM
RENUM
RESTORE
RESUME
RETURN
RIGHT$
RND
RUN
SAVE
SCREEN
SET

SGN
SIN
SOUND
SPACE$
SPC
SPRITE OFF
SPRITE ON
SPRITE STOP
SPRITE$
SQR
STEP
STICK
STOP
STOP ON
STOP OFF
STOP STOP
STRIG
STRIG OFF
STRIG ON
STRIG STOP
STR$
STRING$
SWAP
SWITCH
SWITCH STOP
TAB
TAN
THEN
TIME
TROFF
TRON
USR
VAL
VARPTR
VPEEK
VPOKE
WAIT
WIDTH
XOR

3.6 CONSTANT

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

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

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

Numeric constants are positive or negative numbers. They cannot contain commas. There are six types:

1. Integer
   Whole numbers between -32768 and 32767. Integer constants do not contain decimal points.

2. Fixed-point
   Positive or negative real numbers, i.e., numbers that contain decimal points.

3. Floating-point
   Positive or negative numbers represented in exponential form. A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating-point constant is 10 to 10 .

   For example:

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

   Double precision floating-point constants are denoted by the letter D instead of E.

4. Hex
   Hexadecimal numbers, denoted by the prefix &H.

   For example:

   ```
   &H32F
   ```

5. Octal
   Octal numbers, denoted by the prefix &O.

   For example:

   ```
   &O347
   ```

6. Binary
   Binary numbers, denoted by the prefix &B.

   For example:

   ```
   &B11100111
   ```

3.6.1 Single And Double Precision

Numeric constants may be either single precision or double precision numbers. Single precision numeric constants are stored with 6 digits of precision, and printed with up to 6 digits. Double precision numeric constants are stored with 14 digits of precision and printed with up to 14 digits. Double precision is the default mode.

A single precision constant is any numeric constant that has one of the following characteristics:

1. Exponential form using E.

2. A trailing exclamatiion point (!).

For example:

```
  -1.09E-0622.5!
```

A double precision constant is any numeric constant that has one of these characteristics:

1. Any digits of number without any exponential or type specifier.

2. Exponential form using D.

3. A trailing number sign (#).

For example:
```
3489
345692811
7654321.1234
-1.09432D-06
3489.0#
```

3.7 VARIABLE

Variables are names used to represent values used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a value is assigned to a variable, it is assumed to be zero.

3.7.1 Variable Name and Declaration Character

BASIC variable names may be of any length. Only 2 characters are significant. They can contain letters and numbers. However, the first character must be a letter. Special type declaration characters are also allowed.

A variable name should not be or consists of reserved words. Reserved words include all BASIC commands, statements, function names and operator names. If a variable begins with FN, it is assumed to be a user-defined function.

Variable may represent either a numeric value or a string. String variable names are written with a dollar sign($) as the last character. For example:

```
A$ = "SALES REPORT"
```

The dollar sign is a variable type declaration character. It declares that the variable will represent a string.

Numeric variable names may declare integer, single precision or double precision values. The type declaration characters for these variable names are as follows:

%	Integer variable
!	Single precisin variable
#	Double precision variable

The default type for a numeric variable name is double precision.

Examples of BASIC variable names:

PI#	Declare a double precision value
MINIMUM!	Declare a single precision value
LIMIT%	Declare an integer value
N$	Declare a string value
ABC	Represent a double precision value

Besides, DEFINT, DEFSTR, DEFSNG, and DEFDBL statements may be included in a program to declare the types for certain variable names.

3.7.2 Array Variable

An array is a group or table of values that are referred to with one name. Each individual value in the array is called an element. Array elements are variables.

An array need to be defined and dimensioned. Defining means declaring the name and type of an array.
Dimensioning means setting the number of elements and their arrangement in an array. The maximum number of dimensions for an array is 255.
For example:

```
  DIM A$(3)
```

This creates a one-dimensional array named A$. All its elements are string variables with an initial null value. This array consists of four containers :

A$ (0)
A$ (1)
A$ (2)
A$ (3)

This first string in the list is named A$ (0).

Let's create a two-dimensional array named B, which consists of single-precision variables. All elements are initially set to zero.

```
  DIM B (1,2)
```

COLUMN

ROW	B (0,0)	B (0,1)	B (0,2)
	B (1,0)	B (1,1)	B (1,2)

The element in the second row, first column is called B (1,0)

If an array element is used before the array is dimensioned, it is set to a one-dimensional array with 11 elements.

3.7.3 Space Requirement

The following table lists only the number of bytes occupied by the values represented by the variable names.

Variables	Type	Bytes
	Integer	2
	Single Precision	4
	Double Precision	8
Arrays	Type	Bytes
	Integer	2
	Single Precision	4
	Double Precision	8
Strings	3 bytes overhead plus the present contents of the string.	

3.8 TYPE CONVERSION

A numeric constant can be converted from one type to another. The following rules should be kept in mind.

1. If a numeric variable of one type is set equal to a constant of another type, the number will be stored as the type declared in the variable name.
   For example:

```
10 A%=23.42
20 PRINT A%
RUN
23
Ok
```

   Line 10 specifies the variable A as an integer. It then sets A to be 23.42. However, the type declaration of a variable name takes precedence.

2. As an expression is evaluated, its operands are converted to the same degree of precision. Always, the most precise form is chosen. This is the same for both arithmetic or relational operation. Also, the result of an arithmetic operation is returned in this degree of precision. For example:

```
10 D=6/7
20 PRINT D
RUN
.85714285714286
Ok
```

Calculation was performed in double precision and the result was returned as a double precision value.

```
10 D!=6/7
20 PRINT D!
RUN
.857143
Ok
```

Calculation was performed in double precision and the result was rounded to a single precision value.

3. Logical operators convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

4. When a floating-point value is converted to an integer, the fractional portion is truncated. For example:

```
10 C%=55.88
20 PRINT C%
RUN
55
Ok
```

5. If a double precision variable is assigned a single precision value, only the first six digits of the converted number will be valid. This is because only six digits of accuracy were supplied with the single precision value. For example:

```
10 A!=SQR(2)
20 B=A!
30 PRINT A!,B
RUN
 1.41421      1.41421
Ok
```

3.9 EXPRESSION AND OPERATOR

An expression may be a string, a numeric constant, a variable or a combination of constants and variables with operators which produces a single value.

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

1. Arithmetic
2. Relational
3. Logical
4. Functional

Each category is described in the following sections.

3.9.1 Arithmetic Operator

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, Floating-point Division	X*Y X/Y
+,-	Addition, subtraction	X+Y X-Y

Use parentheses to change this order of precedence. Operations lying within parentheses are performed first. Inside them, the usual order of operations is maintained.

3.9.1.1 Integer Division And Modulus Arithmetic

Two additional operators are available in BASIC:

Integer division is denoted by the "\" symbol. Operands are truncated to integers in the range -32768 to 32767 before division is performed, and the quotient is truncated to an integer. For example:

```
PRINT 10\4
2
Ok
PRINT 25.68\6.99
4
Ok
```

Integer division follows multiplication and floating-point division in order of precedence.

Modulus arithmetic MOD yields the integer value of the remainder of an integer division. For example:

```
PRINT 10.4 MOD 4        10/4=2 has a
2                       remainder 2
Ok
PRINT 25.68 MOD 6.991   25/6=4 has a
1                       remainder 1
Ok
```

Modulus arithmetic follows integer division in order of precedence.

3.9.1.2 Overflow and Division By Zero

As an expression is evaluated, if any value lying beyond the range -32768 to 32767 is encountered, the error message "Overflow" will be displayed. Execution will also be terminated.

If division by zero is encountered, the "Division by zero" error message will be displayed. Likewise, program execution is stopped.

3.9.2 Relational Operator

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow. (See description for "IF" statements.)

The relational operators are:

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

The equal sign is also used to assign a value to a variable.

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

```
X+Y <(T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z.

More examples:

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

3.9.3 Logical Operator

Logical operators perform tests on multiple relations, bit manipulation or Boolean operations. The logical operator returns a bitwise result which is either true (not zero) or flase (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in Table 1. The operators are listed in order of precedence.

Table 3.1 BASIC Logical Operators Truth Table

NOT	X	NOT X	
	1	0	
	0	1	
AND	X	Y	X AND Y
	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR	X	Y	X OR Y
	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR	X	Y	X XOR Y
	1	1	0
	1	0	1
	0	1	1
	0	0	0
EQV	X	Y	X EQV Y
	1	1	1
	1	0	0
	0	1	0
	0	0	1
IMP	X	Y	X IMP Y
	1	1	1
	1	0	0
	0	1	1
	0	0	1

Logical operators can connect two or more relations and return a true or false value to be used in a decision.

Example:

```
IF D > 200 AND F > 4 THEN
80
IF I > 10 OR K > 0 THEN 50
IF NOT (P = -1) THEN 100
```

Logical operators convert their operands to 8-bit, signed, two's complement integers in the range -32768 to 32767. The given operation is performed on these integers in bitwise fashion. Each bit of the result is determined by the corresponding bits in the two operands.

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

63 AND 16 = 16	63	0011	1111
	16	0001	0000
	63 AND 16	0001	0000
15 AND 14 = 14	15	0000	1111
	14	0000	1110
	15 AND 14	0000	1110
-1 AND 8 = 8	-1	1111	1111
	8	0000	1000
	-1 AND 8	0000	1000
4 OR 2 = 6	4	0000	0100
	2	0000	0010
	4 OR 2	0000	0110
10 OR 10 = 10	10	0000	1010
	10	0000	1010
	10 OR 10	0000	1010
-1 OR -2 = -1	-1	1111	1111
	-2	1111	1110
	-1 OR -2	1111	1111
TWOSCOMP = (NOTX) + 1	The two's complement of any integer is the bit complement plus one.		

3.9.4 Functional Operator

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

BASIC also allows user-defined functions that are written by the programmer. See descriptions for DEF FN.

3.9.5 String Operation

Strings may be concatenated by using "+".

Example:

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

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

=	Equality
<> or <>	Inequality
<	Less than
>	Greater than
<= or =<	Less than or equal to
>= or =>	Greater than or equal to

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. Otherwise the precedence is determined according to ASCII codes. If during string comparison the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant.

Examples:

```
"AA"<"AB"
"FILENAME"="FILENAME"
"X&">"X#"
"CL ">"CL"
"Kg"<"KG"
"SMYTH"<"SMYTHE"
B$<"9/12/83"
where B$="8/12/83"
```

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

3.10 PROGRAM EDITING

The Full Screen Editor equipped with BASIC allows the user to enter program lines as usual, then edit an entire screen before recording the changes. This time-saving capability is made possible by special keys for cursor movement, character insertion and deletion, and line or screen erasure. Specific functions and key assignments are discussed in the following sections.

With the Full Screen Editor, a user can move quickly around the screen, making corrections where necessary. The changes are entered by placing the cursor on the first logical line changed and pressing $\boxed{\text{ENTER}}$ at the beginning of each line. A program line is not actually changed unless $\boxed{\text{ENTER}}$ is pressed from somewhere within the logical line.

3.10.1 Writing Program

Within BASIC, the editor is in control any time after an "Ok" prompt and before a RUN command is issued. Any line of text that is entered is processed by the editor. Any line of text that begins with a number is considered as a program statement.

Program statements are processed by the editor in one of the following ways:

1. A new line is added to the program. This occurs if the line number is valid (0 through 65529) and at least one non-blank character follows the line number.

2. An existing line is modified. This occurs if the line number matches that of an existing line in the program. The existing line is replaced with the text of the new line.

3. An existing line is deleted. This occurs if the line number matches that of an existing line, and the new line contains only the line number.

4. An error is produced.

If an attempt is made to delete a non-existent line, an "Undefined line number" error message is displayed.

If program memory is exhausted and a line is added to the program, an "Out of memory" error is displayed and the line is not added.

More than one statement may be placed on a line. If this is done, the statements must be separated by a colon (:). The colon need not be surrounded by spaces.

The maximum number of characters allowed in a program line, including the line number, is 255.

3.10.2 Editing Program

Use the LIST statement to display an entire program or range of lines on the screen so that they can be edited. Text can then be modified by moving the cursor to the place where the change is needed and perform one of the following actions:

1. Typing over existing characters

2. Deleting characters to the right of the cursor

3. Deleting characters to the left of the cursor

4. Inserting characters

5. Appending characters to the end of the logical line

These actions are performed by special keys assigned to the various Full Screen Editor functions (see next section).

Changes to a line are recorded when ENTER is pressed while the cursor is somewhere on the line. All changes for that logical line are entered, no matter how many physical lines are included.

3.10.3 Full Screen Editor

The following table lists the hexadecimal codes for the BASIC control characters and summarizes their functions. The Control-key sequence normally assigned to each function is also listed. These conform as closely as possible to ASCII standard conversions.

Individual control functions are described in the following the table.

Table 3.2 SV BASIC Control Functions. The ASCII control key is entered by pressing the key while holding down the Control key.

HEX Code	Control Key	Special Key	Function	Remark
01	A		Ignored	
02	B		Move cursor to start of previous word	The cursor is moved left to the previous word. The previous word is defined as the next character to the left of the cursor in the sets A-Z, a-z or 0-9.
03	C		Break when BASIC is waiting for input	Return to BASIC direct mode, without saving changes that were made to the line currently being edited.
04	D		Ignored	
05	E		Truncate line (clear text to end of logical line)	Move cursor to end of logical line. Delete the characters passed over. Characters typed from the new cursor position are appended to the line.
06	F		Move cursor to start of next word	Next word is the next character to the right of the cursor in the sets A-Z, a-z or 0-9.
07	G		Beep	Produce the beep sound.
08	H	⇦	Backspace, deleting characters passed over	Delete the character to the left of the cursor. All characters to the right of the cursor are moved left one position. Subsequent characters and lines within the current logical line are moved up (wrapped).

09	I		Tab (moves to next TAB stop)	Move cursor to the next tab stop, overwriting blanks. Tab stops occur every 8 characters.
0A	J		Line feed	
0B	K	CLS/HM	Move cursor to home position	Move cursor to the upper left corner of the screen. The screen is not blanked.
0C	L	CLS	Clear screen	Move cursor to home position and clear the entire screen, regardless of where the cursor is positioned when the key is entered.
0D	M	ENTER	Enter (enter current logical line)	End the logical line and return to BASIC.
0E	N		Append to end of line	Move cursor to end of line, without deleting the characters passed over. All characters typed from the new position until an ENTER are appended to the logical line.
0F	O		Ignored	
10	P		Ignored	
11	Q		Ignored	
12	R	INS	Toggle insert/typeover mode	Toggle switch for insert mode. When insert mode is on, the size of the cursor is reduced and characters are inserted at the current cursor position. Characters to the right of the cursor move right as new ones are inserted. Line wrap is observed.
13	S		Ignored	

14	T		Ignored	
15	U		Clear logical line	When the key is entered anywhere in the line, the entire logical line is erased.
16	V		Ignored	
17	W	SELECT	Ignored	
18	X		Ignored	
19	Y		Ignored	
1A	Z		Ignored	
1B	[	ESC	Ignored	
1C	\	→	Cursor right	Move cursor one position to the right. Line wrap is observed.
1D	]	←	Cursor left	Move cursor one position to the left. Line wrap is observed.
1E	^	↑	Cursor up	Move the cursor up one physical line (at the current position).
1F	-	↓	Cursor down	Move the cursor down one physical line (at the current position).
7F	DEL	DEL	Delete character at cursor	

Normally, a logical line consists of all the characters on its physical lines. During execution of an INPUT or LINE INPUT statement, however, this definition is modified slightly to allow for forms input. The logical line is restricted to characters actually typed or passed over by the cursor. Insert mode and delete function only move characters within a logical line.

Insert mode increments the logical line except when the characters moved will write over non-blank characters that are on the same physical line but not part of the logical line. In this case, the non-blank characters not being part of the logical line are preserved and the characters at end of the logical line are thrown out. This preserves labels that existed prior to the INPUT statement. If an incorrect character is entered as a line is being typed, it can be deleted with the Back Space key (⇦) or with Control-U. This simply backspaces over a character and erases it. Once a character has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, typed Control-H.

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

To delete the entire program currently residing in memory, enter the NEW command. NEW is usually used to clear memory prior to entering a new program.

3.11 SPECIAL KEY

BASIC supports several special keys as follows.

3.11.1 Function Key

BASIC has 10 pre-defined function keys. The current contents of these keys are displayed on the last line on the screen and can be redefined by program with KEY statement. The initial values are:

F1	color
F2	auto[10,10] $\boxed{\text{ENTER}}$
F3	goto
F4	list
F5	run $\boxed{\text{ENTER}}$
F6	color 15,4,5 $\boxed{\text{ENTER}}$
F7	cload"
F8	cont $\boxed{\text{ENTER}}$
F9	list. $\boxed{\text{ENTER}}$
F10	[CLS]run $\boxed{\text{ENTER}}$

Function keys are also used as event trap keys. See ON KEY GOSUB and KEY ON/OFF/STOP statement for details.

3.11.2 Stop Key

When BASIC is in command mode, the STOP key has no effect to the operation.

When BASIC is executing the program, pressing the STOP key causes suspension of the program execution. BASIC turns on cursor display to indicate that the execution is suspended. Another STOP key input resumes the execution. If the STOP key and control key are pressed simultaneously, BASIC terminates the execution and return to command mode with the following message.

```
Break in<nnnnn>
```

where<nnnnn>is the program line number where the execution stopped.

3.12 ERROR MESSAGES

If an error causes program execution to terminate, an error message is printed. For a complete list of BASIC error codes and error messages, see Appendix A.

3.13 INPUT AND OUTPUT

3.13.1 DATA FILES

A file is a collection of information, kept somewhere other than in the random access memory. This may be tape or diskette.

There are two categories of data files, namely, sequential and random access file.

In order to keep such file orderly, two criteria should be specified: file number and filename.

3.13.1.1 File Number

Filenumber is what the computer uses to refer to the file. It is a unique number that is associated with the physical file that is opened. This identifies the route that the computer uses to send and receive information from the specific device.

3.13.1.2 Naming File

The physical file is specified by its file specification, which is a string expression of the form:

```
device:  filename
```

The device name tells BASIC where to seek for the file and the filename tells BASIC which file to look for on that device. Sometimes it is not necessary to specify these items. Take for example, you want to retrieve the first file from the cassette, both file number and filename may be omitted.

The colon (:) is part of the device name. Whenever a device is stated, you must include the colon, even if the filename is not given.

Remember to enclose the file specification with a pair of quotation marks. For example:

```
LOAD "  device  :  filename  "
```

3.13.1.2.1 Device Name

A maximum of four characters can be used, including the colon (:).

KYBD:	Keyboard. Input only. For cassette and disk BASIC.
CRT :	Screen. Output only. For cassette and disk BASIC.
LPT :	Printer. Output only. For cassette and disk BASIC.
CAS :	Cassette tape player. As a storage device. Input and output. For cassette and disk BASIC.
1 :	First disk drive. Input and output. For cassette and disk BASIC.
2 :	Second disk drive. Input and output. For cassette and disk BASIC.

3.13.1.2.2 Filename:

The filename must conform to the following rules.

For cassette files: The name is restricted to a length of six characters.

For disk files: The name may consist of two parts seperated by a period (.):

```
        name. extension
```

The maximum number of characters for name is six and that for extension is three. Extra character will be truncated.

If a decimal point appears in a filename after fewer than six characters, the name is blank filled to the sixth character, and the next three characters are the extension. If the filename has more than six characters, BASIC automatically inserts a decimal point after the sixth character and uses the next three as an extension. Extra is ignored.

3.13.2 SCREEN

The TMS9918/9929 Video Display Processor supports the text and graphic display on the screen such as text, special characters, points, lines or more complex shapes in color or in black and white.

Text refers to alphabets, numbers and all the special characters in the regular character set.

Enjoy also your computer's capability in drawing pictures with the special line and box characters. Do not miss out the ingenious SPRITE. You may also create blinking, reverse, invisible and highlighted image with the help of BASIC commands. In both high and low resolution screen, all points are addressable. The screen can be divided into three layers, one lying on top of the other. Starting from the bottom, they are the border, the background and the foreground.

A total of sixteen colors can be displayed. Each one is characterised by a number.

COLOR #	COLOR
0	Transparent
1	Black
2	Medium Green
3	Light Green
4	Dark Blue
5	Light Blue
6	Dark Red
7	Cyan
8	Medium Red
9	Light Red
10	Dark Yellow
11	Light Yellow
12	Dark Green
13	Magenta
14	Gray
15	White

The colors may vary depending on your display device. Adjusting the color tuning may help set the colors to match this chart better.

3.13.2.1 Text Mode

This is the default mode as the display screen is first turned on. Or else set by the command (SCREEN 0). Here you can communicate with the computer through keyboard input.

In this mode, the background covers the border totally. The foreground carries all the images that appear on the screen, ie. the text.

Characters are shown in 24 horizontal lines across the screen, numbered 1 through 24, from top to bottom. Each line has 39, 40 or 80 characters. The default is 39 without 80 column cartridge; or 80 if the latter is installed. With the command WIDTH 40, a maximum of 40 characters can be displayed. The numbering starts at 1 from left to right. The position numbers are used in the following commands or functions:

LOCATE	POS	CSRLIN

The top left corner of the screen has the coordinate of (1,1). Use PRINT statement to place any desired characters on the screen. The character is displayed at the position of the cursor. If you command a string of characters to be PRINTed, they will be printed from left to right on a line. When the cursor will normally go to the 24th line, lines 1 through 23 are scrolled up one line, so that what was line 1 disappears from the screen. Line 24 is then blanked, and the cursor remains on line 23 to continue printing.

Line 24 is usually used to display the current function keys. It is however possible to write over this line.

The useful commands or functions you can use to display information in text mode are:

CLS	POS	TAB
COLOR	PRINT	WIDTH
CSRLIN	SCREEN	WRITE
LOCATE	SPC	

3.13.2.2 Graphics Mode

There are two graphic resolution available. Here are the useful commands or functions to generate an impressive picture:

CIRCLE	LINE	PSET
COLOR	PAINT	PUT
DRAW	POINT	SCREEN
GET	PRESET	

High Resolution

This is set by the command SCREEN 1. There are 256 horizontal points (or pixels) and 192 vertical points. These points are numbered from left to right and from top to bottom. The top left corner has the coordinate (0,0). Text characters can be displayed in this graphic mode. The size of character is the same as in the text mode.

Low Resolution

This is set by the command SCREEN 2. There are 64 horizontal points and 48 vertical points. The numbering is similar as the high resolution mode.

INPUT/OUTPUT

Any type of input/output may be treated like I/O to a file.

3.13.3.1 Sound and Music

You can use the followings to create sound on the computer.

BEEP	Beep the speaker.
SOUND	Make a single sound of desired frequency and duration.
PLAY	Play music as indicated by a character string.

3.13.3.2 Joystick

Joystick is useful in an interactive environment. BASIC supports joystick and graphic tablet. Refer to the following commands or functions for details:

PAD   STICK   STRIG

CHAPTER 4

BASIC COMMANDS, STATEMENTS AND FUNCTIONS

4.1 COMMANDS, STATEMENTS AND FUNCTIONS EXPECT I/O

4.1.1 Commands except I/O

4.1.1.1 AUTO

Purpose :	To generate a line number automatically after pressing ENTER.
Version :	Cassette, Disk
Format :	AUTO [line number [, increment]]
Remarks :	AUTO begins numbering at line number and increments each subsequent line number by increment. The default for both value is 10. If line number is followed by a comma but increment is not specified, the last increment specified in an AUTO command is assumed.

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

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

Example :

```
   AUTO
```

Generate line numbers 10, 20, 30,....

```
   AUTO 20, 5
```

Generate line numbers 20, 25, 30,....

```
   AUTO 100,
```

Generate line numbers 100,105,110,.... The increment is 5 since the previous AUTO command has specified the increment to be 5.

```
   AUTO,3
```

Generate line numbers 0,3,6,....

4.1.1.2 CLEAR

Purpose : To set all numeric variables to zero, all string variables to null, close all open files end optionally, to set the end of memory.

Version : Cassette, Disk

Format : CLEAR [string space [, highest location]]

Remarks : string space specifies the space for string variables. This is useful to reserve space in storage for machine language programs. Default size is 200 bytes. highest location sets the highest memory location available for use by BASIC.

CLEAR frees all memory used for data without erasing the program which is currently in memory. Also, arrays are undefined; numeric variables are set to zero; string variables are nullified; any information set with DEF statement is lost.

Example :

```
  CLEAR
```

Clear all data from memory without erasing the program.

```
  CLEAR 32768
```

Clear the data and set the maximum workspace size to 32K bytes.

```
  CLEAR 32768, 1000
```

Clear data; set the maximum workspace for BASIC to 32K bytes; set the stack size to 1000 bytes.

4.1.1.3 CLICK

Purpose : Turn on/off the keyboard click sound.

Version : Cassette, Disk

Format : CLICK ON/OFF

Remarks : Pressing a key is echoed with a "click" sound.

Example :

```
10 CLICK ON
20 INPUT "TYPE IN A SENTENCE, THEN
   PRESS ENTER"; S1$
40 INPUT "TYPE THE SAME SENTENCE
   AGAIN AND NOTE THE DIFFERENCE";
   S2$
```

As the first sentence is typed, the clicking sound is heard. This is not so as the second sentence is input.

4.1.1.4 CONT

Purpose : To continue program execution after CONTROL + C keys being pressed simultaneously, or a STOP or END statement has been executed.

Version : Cassette, Disk

Format : CONT

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

CONT is usually used with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO which resumes execution at a specified line number. Also it may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break. Execution cannot be continued if a direct mode error has occurred during the break.

Example : Create a loop. During program execution, interrupt it by pressing CTRL + STOP keys simultaneously.

```
10 FOR I = 1 TO 9
20 PRINT I;
30 NEXT
40 PRINT
50 GOTO 10
RUN
1  2  3  4  5  6  7  8  9
1  2  3  4  5  C
Break in 20
Ok
CONT
6  7  8  9
1  2  3  4  5  6  7  8  9
•
•
•
•
```

4.1.1.5 DATA

Purpose	: To store the numeric and string constants that are accessed by the program's READ statement(s).
Version	: Cassette, Disk
Format	: DATA list of constants

Remarks : list of constants may contain numeric constants in any format; i.e., fixed point floating point or integer. No numeric expressions such as 1/4, 2*3 are allowed in the list. String constants in DATA statements must be surrounded by double quotation marks only if they contain comma, colons, or leading or trailing blank. Otherwise, quotation marks are not needed.

DATA statements are nonexecutable and may be placed anywhere in the program. It may contain as many constants as will fit on a logical line constants are separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order of line numbers. They may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement. DATA statements may be read from the beginning or specified line by use of the RESTORE statement.

Examples :

```
10 FOR I = 1 TO 3
20 READ NAM$(I), AGE(I)
30 NEXT
40 DATA JOHN, 42, JOSEPHINE, 24,
   LEO, 21
```

This program reads string and numeric data from the DATA statement in line 40. If a colon follows the name, line 40 will be changed to

```
40 DATA "JOHN:", 42, "JOSEPHINE:",
   24, "LEO:", 21
```

4.1.1.6 DEF FN

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

Version : Cassette, Disk

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

Remarks : name must be a legal variable name. This name, preceded by FN, becomes the name of the function. parameter list is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. function definition is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a 'Type mismatch' error occurs.

A DEFFN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an 'Undefined user function' error occurs. DEFFN is illegal in the direct mode.

Example :

```
10 DEF FNAREA (B,H) = B * H/2
20 INPUT "BASE ="; BASE
30 INPUT "HEIGHT ="; HEIGHT
40 PRINT "AREA IS" FNAREA
   (BASE, HEIGHT)
RUN
BASE = ? 3
HEIGHT = ? 6
AREA IS  9
Ok
```

Line 10 defines the function FNAREA. The function is called and then printed in line 40.

4.1.1.7 DEFUSR

Purpose : To specify the starting address of an assembly language subroutine, which is called by the USR function.

Version : Cassette, Disk

Format : DEFUSR[digit]= integer expression

Remarks : digit may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If digit is omitted, DEFUSRO is assumed. The value of integer expression is the starting address of value of of the USR routine.

Any number of DEFUSR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example :

```
100 DEF USRO = 1000
200 X = USRO (Y * 5)
```

This example calls a routine at absolute location 1000 in memory.

4.1.1.8 DEFINT DEFSNG DEFDBL DEFSTR

Purpose : To declare variable type as integer, single precision, double precision or string.

Version : Cassette, Disk

Format :

```
DEFINT  range(s) of letters
DEFSNG  range(s) of letters
DEFDBL  range(s) of letters
DEFSTR  range(s) of letters
```

Remarks : DEFINT/SNG/DBL/STR statements specify the variable types to be integer variable/single-precision variable/double-precision variable/ string variable. However, a type declaration character (%,!,# or $) always takes precedence over a DEFxxx statement in the type of variables. (See the end of section 3.6 for details of declaration characters.)

Example :

```
10 DEFINT A, B
20 DEFSNG E, F
30 DEFDBL D, G-I
40 DEFSTR C
50 AVERAGE = (2+3+6)/3 : PRINT
   AVERAGE
60 EVASAVG = (1+3+1)/3 : PRINT
   EVASAVG
70 DANSAVG = (2+4+1)/3 : PRINT
   DANSAVG
80 COMMENT = "AVERAGE IS LOW" :
   PRINT
   COMMENT
RUN
3
1.66667
2.3333333333333
AVERAGE IS LOW
Ok
```

Line 10 declares that all variables beginning with the letter A or B will be integer variable.

Line 20 renders all variables beginning with letter E or F to be single-precision variables.

Line 30 declares that all variables beginning with letter D, G, H or I will be double-precision variables.

Line 40 causes all variables beginning with letter C to be string variables.

4.1.1.9 DELETE

Purpose : To delete program lines.

Version : Cassette, Disk

Format : DELETE [line number] [- line number]

Remarks : The beginning line number is the first line to be deleted. The ending line number is the last line to be deleted.

A period (.) may be used in place of the number to indicate the current line. If no line number is specified, an "Illegal function call" error occurs.

BASIC always returns to command level after a DELETE command is executed.

Example : Suppose the following program is entered.

```
10   FOR H = 0 TO 23
20   FOR M = 0 TO 59
30   FOR S = 0 TO 59
40   CLS
50   PRINT H ":" M ":" S
60   BEEP
70   FOR T = 1 TO 50
80   NEXT T
90   NEXT S
100  NEXT M
110  NEXT H
```

```
DELETE 10
```

Line 10 is deleted.

```
   DELETE .
```
Line 110 is deleted.

```
   DELETE 60-80
```
Lines 60, 70, 80 are deleted.

```
   DELETE -100
```
Lines 20, 30, 40, 50 90 and 100 are deleted.

4.1.1.10 DIM

Purpose	: To specify the maximum values for array variable subscripts and allocate storage accordingly.
Version	: Cassette, Disk
Format	: DIM<list of subscripted variables>
Remarks	: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range " error occurs. The minimum value for a subscript is always 0. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767. Yet both numbers are limited by memory size and statement length.
	An array can only be dimensioned once. Otherwise use the ERASE statement to erase an array for redimensioning.

Example : The following example creates two arrays: a one- dimensional string array named M$ with 13 elements, M$(0) through M$(12); and a numeric array named D with 13 elements D(0) through D(12).

```
10 DIM M$(12)
20 DIM D(12)
30 PRINT "YEAR" TAB(5) 1984
40 FOR I = 1 TO 12
50 READ M$(I), D(I)
60 PRINT M$(I) TAB(6) D(I)
70 NEXT
80 DATA JAN, 31, FEB, 29, MAR, 31,
   APR, 30, MAY, 31, JUN, 30, JUL,
   31,
   AUG, 31, SEP, 30, OCT, 31, NOV,
   30,
   DEC, 31

RUN
YEAR  1984
JAN    31
FEB    29
MAR    31
APR    30
MAY    31
JUN    30
JUL    31
AUG    31
SEP    30
OCT    31
NOV    30
DEC    31
Ok
```

Now Change the program to read:

```
10 DIM A $ (12, 1)
20 FOR J = 0 TO 12
30 FOR K = 0 TO 1
40 READ A$ (J,K),
50 PRINT A$ (J,K)
60 NEXT K
70 PRINT
80 NEXT J
90 DATA  YEAR, 1984, JAN, 31,
   FEB, 29,
   MAR, 31, APR, 30, MAY, 31, JUN,
   30,
   JUL, 31, AUG, 31, SEP, 30, OCT,
   31,
   NOV, 30, DEC, 31
```

In this example, a two-dimensional array A$ with 26 elements, A$(0,0) through A$(12,1) is created.

4.1.1.11 END

Purpose : To terminate program execution, close all files and return to command level.

Version : Cassette, Disk

Format : END

Remarks : END statement may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional.

Example :

```
10 READ X
20 PRINT X
30 IF X   100 THEN END ELSE GOTO
   10
40 DATA 50, 200
RUN
50
200
Ok
```

Per line 30, the program will terminate if value of X exceeds 100.

4.1.1.12 ERASE

Purpose : To eliminate arrays from a program.

Version : Cassette, Disk

Format : ERASE   list of array variables

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

Example :

```
10 PRINT FRE(0);
20 DIM A (50, 50)
30 PRINT FRE (0);
40 ERASE A
50 DIM A (10, 10)
60 PRINT FRE (0)
RUN
29126 8308 28148
Ok
```

This example uses the FRE function to illustrate how ERASE can be used to free memory. If dimensioned as A(50,50) 20K bytes of memory (29126-8308) is required. When dimensioned as A(10,10), less memory 1K bytes (29126-28148) is required.

Type
```
   DELETE 40
```
Run the program to see what happens.

4.1.1.13 ERROR

Purpose : To simulate the occurrence of an error or to allow error codes to be defined by the user.

Version : Cassette, Disk

Format : ERROR integer expression

The value of integer expression must be greater than 0 and less than 255. If the value of integer expression equals an error code already in use by BASIC, the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed.

To define your own error code, use a value that is greater than any used by BASIC for error codes. See Appendix A for a list of error codes and messages. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to BASIC.) This user defined error code may then be conveniently handled in an error trap routine.

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

Example : In direct mode:

```
   ERROR 10
   Undefined array
   Ok
```

Or define error message.

```
   10 READ A$
   20 IF A$ = "FALSE" THEN ERROR 250
   30 DATA FALSE
   RUN
   Unprintable error in 20
   Ok
```

4.1.1.14 FOR ..... NEXT

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

Version : Cassette, Disk

Format : FOR variable =x To y [STEP z]

Remarks : variable can be integer, single-precision or double-precision, where x, y, z are numeric expressions.

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

The body of the loop is executed one time at least if the initial value of the loop times the sign of the step is less than the final value times the sign of the step.

FOR.....NEXT loops may be nested, that is, a FOR..... NEXT loop may be placed within the context of another FOR .....NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them. Such nesting of FOR ..... NEXT loops is limited only by available memory.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a 'NEXT without FOR' error message is issued and execution is terminated.

Example :

```
10 SUM = 0
20 FOR X = 1 TO 100
30 SUM = SUM + X
40 NEXT
50 PRINT "THE SUM OF INTEGERS FROM
1 TO 100 is" SUM
RUN
THE SUM OF INTEGERS FROM 1 TO 100
is 5050
Ok
```

The loop is executed a hundred times, starting from 1 to 100 with an increment of 1.

```
10 SCREEN 2
20 FOR X = 0 TO 20 STEP 2
30 FOR Y = 20 TO 0 STEP -2
40 PSET (X, Y)
50 NEXT Y
60 NEXT X
```

Nested loops are created in the second example. The increment for X is 2. The decrement for Y is 2. Loop Y is executed 11 times before loop X is executed once.

4.1.1.15 GOSUB ..... RETURN

Purpose : To branch to subroutine beginning at line number and return from a subroutine.

Version : Cassette, Disk

Format : RETURN [<line number>]

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

The RETURN statement(s) in a subroutine causes BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, it may be preceded by a STOP, END or GOTO statement that directs program control around the subroutine. Otherwise a 'RETURN without GOSUB' error message is issued and execution is terminated.

Example :

```
10   INPUT "WHAT'S THE DAY OF THE
     WEEK"  ; A
20   GOSUB 60
30   INPUT "MEMO"; M $
40   GOTO 10
50   IF A=1 THEN PRINT "MONDAY"
60   UF A=2 THEN PRINT "TUESDAY"
70   IF A=3 THEN PRINT "WEDNESDAY"
80   IF A=4 THEN PRINT "THURSDAY"
90   IF A=5 THEN PRINT "FRIDAY"
100  IF A=6 THEN PRINT "SATURDAY"
     ELSE PRINT
     "SUNDAY"
110  RETURN
```

This shows how a subroutine works. The GOSUB in line 20 calls the subroutine starting at line 60. So the program branches to line 60 and start excuting statements there until it strikes the RETURN statement in line 110. The program is sent back to the statement after the subroutine call, i.e., line 30. The GOTO statement in line 50 prevents the subroutine from being performed the second time.

4.1.1.16 GOTO

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

Version : Cassette, Disk

Format : GOTO <line number>

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

Example :

```
100 PRINT "DO YOU WANT ANOTHER
    GAME (Y/N)"
200 A$ = INKEY$
300 IF A$ < >CHR$(89) AND A$
    CHR$(78) THEN GOTO 200
400 IF A$ = CHR$(89) THEN GOTO 10
500 END
```

This shows how a program is branched. Line 300 does a checking first, then branches the program to line 200 if the result is positive. Likewise for line 400.

4.1.1.17 IF ..... THEN ..... ELSE IF ..... GOTO ..... ELSE

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

Version : Cassette, Disk

Format : IF < expression > THEN <statement(s)> / < line number > [ELSE <statement(s) >/ < line number >]
IF < expression > GOTO <line number> [ELSE <statement(s) > / < line number >]

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

IF.....THEN.....ELSE statements may be nested. Nesting is limited only by the length of the line.
If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN.

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

The computer will not print "A< >C" when A< >B.

It will print "A< >C" when A=B and B<>C.

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

```
   IF 2 + 2 > 2 THEN PRINT "2 + 2 IS
   LARGER THAN 2"
   2 + 2 IS LARGER THAN 2
   Ok
```

Since 2 + 2 > 2 is a true statement, the THEN clause is executed.

```
   IF 2>3 THEN PRINT "2 IS LESS THAN
   3"
   ELSE PRINT "2>3 IS FALSE"
   2>3 IS FALSE
   Ok
```

Since 2 > 3 is false the ELSE clause is executed.

```
10 PRINT "HEAD OR TAIL (0 OR 1)"
20 FOR I = 1 TO 5
30 PRINT I "IS";
40 INPUT N
50 IF N   0 AND N   1 GOTO 40
60 T = T + N
70 NEXT
80 PRINT "OUT OF 5 TRIALS, THERE
   ARE "5 - T "HEADS AND " T
   "TAILS"
RUN
HEAD OR TAIL (0 OR 1)
1 IS? P
? REDO FROM START
? 0
2 IS? 1
3 IS? 1
4 IS? 1
5 IS? 0
OUT OF 5 TRAILS, THERE ARE 2 HEADS
AND 3 TAILS
Ok
```

Line 50 tests whether 0 or 1 is entered. If not, the program flow is directed to line 40.

4.1.1.18 INPUT

Purpose : To allow input from the keyboard during proogram execution.

Version : Cassette, Disk

Format : INPUT ["<prompt string>";]<list of variables>

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

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

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

Responding to input with the wrong type of value (string instead of number, etc.) causes the message '?Redo from start' to be printed. No assignment of input value is made until an acceptable response is given.

Example :

```
10 INPUT "A and B";A,B
20 PRINT A+B
Ok
run
A and B? 10, E
?Redo from start
A and B? 10,20
30
Ok
```

Responding to INPUT with too many items causes the message "?Extra ignored" to be printed and the next statement to be executed.

```
run
A and B? 10,20,30
?Extra ignored
30
Ok
```

Escape INPUT by typing CTRL-C or the CTRL and STOP keys simultaneously. BASIC returns to command level and types "Ok". Typing CONT resumes execution at the INPUT statement.

4.1.1.19 LET

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

Version : Cassette, Disk

Format : [LET] variable = expression

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

Example :

```
10 LET D1=3
20 LET D2=4
30 LET D3=5
40 SUM = D1+D2+D3
50 PRINT SUM
RUN
12
Ok
```

Try out the above program by deleting LET from lines 10 through 30.

4.1.1.20 LINE INPUT

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

Version : Cassette, Disk

Format : LINE INPUT [" prompt string ";] string variable

Remarks : The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to an ENTER is assigned to string variable

Escape LINE INPUT by typing CTRL-C or the CTRL and STOP keys simultaneously. BASIC returns to command level and types "Ok". Typing CONT resumes execution at the LINE INPUT statements.

Example :

```
LINE INPUT "NAME:";N$
NAME: JOHN K. LIVINGSTONE
Ok
```

The computer prompts you to input a string after the printout "NAME:". Unless ENTER is pressed, the Ok prompt won't display.

4.1.1.21 LIST

Purpose : To list all or part of the program.

Version : Cassette, Disk

Format : LIST [line number [-[line number]]]

Remarks : line number lies in the range 0 to 65529. The beginning line number is the firrst line to be listed. The ending line number is the last line to be listed. If only the first line number is specified, that line is listed.

If "-" and the second line number are specified, all lines from the beginning of the program through that line are listed.

If both line numbers are omitted, the entire program is listed.

A period (.) may be used to indicate the current line number.

Listing is terminated by pressing CTRL + STOP Keys simultaneously. Listing is suspended by depressing STOP. Resume by pressing STOP the second time.

Example : Type in the following program first:

```
10    REM DEMONSTRATION PROGRAM
20    DEFINT A - Z
30    J = 4:  A = 16:  B = 80:  S =
      8
40    ZZ = RND (- TIME)
50    SCREEN 1
60    FOR K = A TO B STEP S
70    C = INT (RND (1) *16)
80    LINE (K,K) - (255 - K, 191 -
      K), C, BF
90    C2 = INT (RND (1) *16)
100   IF C = C2 THEN 90 ELSE COLOR
      C2
110   FOR I = K TO 255 - K STEP J
120   LINE (I,K) - (255 - I, 191 -
      K)
130   NEXT
140   FOR I = 191 - K TO K STEP -J
150   LINE (K,I) - (255 - K, 191 -
      I)
160   NEXT
170   FOR Z = 1 TO 1000:  NEXT:
      NEXT
180   FOR Z = 0 TO 1000:  NEXT
190   SWAP A,B
200   S = -S
210   GOTO 50
```

```
   LIST
```
The entire program is listed on the screen.

```
   LIST 10
```
List line 10.

```
   LIST 10 - 30
```
Lines 10 through 30 and listed.

```
   LIST - 100
```
List from the first line i.e., the lowest line number, through line 100.

4.1.1.22 LLIST

Purpose : To list all or part of the program on the printer.

Version : Cassette, Disk

Format : LLIST [line number [-[line number]]]

Remarks : Refer to LIST command for details.

Example : Refer to LIST command for details.

4.1.1.23 LPRINT LPRINT USING

Purpose : To print data at the line printer.

Version : Cassette, Disk

Format : LPRINT[<list of expression>]
LPRINT USING <string expression> ;
<list of expressions>

Remarks : <list of expression> is a list of numeric and/or string expressions. Expressions should be separated by commas or semicolons.

string expression is a string constant or variable which identifies the format to be used for printing. LPRINT assumes an 132-character wide printer. Thus, BASIC automatically inserts an ENTER or line feed after printing 132 characters. This will result in one line being skipped when 132 characters are printed; unless the LPRINT statement ends with a semicolon.

Example : Refer to PRINT and PRINT USING.

4.1.1.24 MID$

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

Version : Cassette, Disk

Format : MID$(<string expression 1>),n[,m]= <string expression 2>

Remarks : The character in < string expression 1> , beginning at postion n, are replaced by the characters in < string expression 2 >. The optional m refers to the number of characters from < string expression 2 > that will be used in the replacement. Whether m is omitted or included, the replacement of characters never goes beyond the original length of string expression 1 .

Example :

```
10 A$="SCAN"
20 MID$(A$,2,3)="TAR"
30 PRINT A$
RUN
 STAR
Ok
```

On line 20, the second, third and fourth characters of A$ are replaced by the string "TAR".

4.1.1.25 NEW

Purpose	:	To delete entire program from working memory and reset all variables.
Version	:	Cassette, Disk
Format	:	NEW
Remarks	:	It causes all files to be closed and turns trace off if it was on.

NEW is usually used to free memory before entering a new program. BASIC always returns to command level after NEW is executed.

4.1.1.26 ON ERROR GOTO

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

Version : Cassette, Disk

Format : ON ERROR GOTO line number

Remarks : Once error trapping has been enabled, all errors detected, including direct mode errors (e.g., syntax error), will cause a jump to the specified error handling subroutine. If line number does not exist, an "Undefined line number" error results. To disable error trapping, execute an "ON ERROR GOTO 0". Subsequent errors will print an error message and halt execution. An "ON ERROR GOTO 0" statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines, execute an "ON ERROR GOTO 0" if an error is encountered for which there is no recovery action.

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

Example :

```
10  REM GUESS A NUMBER FROM 1 TO
    100
20  ON ERROR GOTO 80
30  A = 1 + INT(100* RND (-TIME))
40  INPUT "YOUR GUESS";B
50  IF A  B THEN ERROR 80 ELSE IF
    A  B THEN ERROR 81
60  PRINT "YOU'VE GOT IT"
70  END
80  IF ERR = 80 THEN PRINT "TOO
    LARGE" ELSE PRINT "TOO SMALL"
90  RESUME 40
100 END
```

This program is a number guessing game. By using error codes 80 and 81 which BASIC doesn't use, the program traps the error if the guessed number, B does not equal to A.

4.1.1.27 ON ... GOTO
ON ... GOSUB

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

Version : Cassette, Disk

Format : ON < expression > GOTO < line number [,< line number >].....
ON < expression > GOSUB < line number [,< line number >].....

Remarks : The value of < expression > determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a noninteger, the fractional portion is discarded.)

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

If the value of < expression > is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of < expression > is negative or greater than 255, an 'Illegal function call' error occurs.

Example :

```
100 ON L GOTO 150, 200, 300
```

If L equals 1, branch to line 150; if L equals 2, branch to 200; if L=3, branch to 300.

```
100 ON M GOSUB 1000, 1500
```

If M equals 1, branch to subroutine starting at line 1000; if M equals 2, branch to subroutine starting at line 1500.

4.1.1.28 POKE

Purpose : To write a byte into a memory location.

Version : Cassette, Disk

Format : POKE <address of the memory> , <integer expression >

Remarks : <address of the memory> is the address of the memory location to be POKEd. The < integer expression> is the data (byte) to be POKEd. It must be in the range 0 to 255. And address of the memory must be in the range -32768 to 65535. If this value is negative, address of the memory location is computed as a subtracting from 65536. For example, -1 is same as the 65535 (=65536-1). Otherwise, an "Overflow" error occurs.

Example :

```
   Poke &H5A00, &HFF
```

Write the data &HFF at the location &H5A00.

4.1.1.29 PRINT

Purpose	:	To output data to the console.
Version	:	Cassette, Disk
Format	:	PRINT [<list of expressions>]

Remarks : If < list of expressions> is included, the values of the expressions are printed at the console. An expression in the list may be a numeric and/or string expression. String must be enclosed in quotation marks.

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each. In the < list of expression > , a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the < list of expressions> , the next PRINT statement begins printing on the same line, spacing accordingly. If the < list of expressions > terminates without a comma or a semicolon, a line feed follows. If the printed line is longer than the screen width, BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign.

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

Example :

```
10 PRINT "AREA OF CIRCLE = 3.1416*
   RADIUS   2"
20 PRINT
30 ? "RADIUS"
40 INPUT R
50 A = 3.1416* R * 2
60 PRINT
70 PRINT "AREA EQUALS TO"; A
RUN
AREA OF CIRCLE = 3.1416* RADIUS*
2
?4
AREA EQUALS TO 50.2656
Ok
```

Line 10 commands a string to be printed. Line 20 renders a line being skipped. Question mark (?) can be used to substitute the word "PRINT". On line 70, both string and number are printed, with no line skipped between both printouts, because of the usage of semicolon (;).

4.1.1.30 PRINT USING

Purpose : To print strings or numerics using a specified format.

Version : Cassette, Disk

Format : PRINT USING <string expression> ; <list of expressions>

Remarks : <list of expressions> comprises the string expressions or numeric expressions that are to be printed, seperated by semicolons. < string expression> is a string literal (or variable comprising special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!"

Specifies that only the first character in the given string is to be printed.

Example:

```
A$="Japan"
Ok
PRINT USING "!";A$
J
Ok
```

" < n spaces > "

Specifies that 2+n characters from the string are to be printed.

If the " " signs are typed with no spaces, two characters will be printed; with one space three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example:

```
A$="Japan"
Ok
PRINT USING "\\";A$
Japa
Ok
```

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

"#"

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

Example:
```
PRINT USING "###.##";10.2, 2, 3.456,
.24, 123.5 10.20 2.00 3.45 0.24
123.50
Ok
```

"+"

A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

Example:
```
PRINT USING "+###.##";1.25,-1.25
  +1.25  -1.25
Ok
PRINT USING "###.##+";1.25,-1.25
  1.25+ 1.25-
Ok
```

"-"

A minus sign at the end of the format will cause negative numbers to be printed with a trailing minus sign.

Example:
```
PRINT USING "###.##-";1.25,-1.25
  1.25  1.25-
Ok
```

"**"

A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two or more digits.

```
Example:
PRINT USING "**#.##";1.25,-1.25
**1.25*-1.25
Ok
```

"$$"

A double dollar sign causes dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the $ sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
Example:
PRINT USING "$$###.##";12.35,-12.35
  $12.35 -$12.35
Ok
PRINT USING "$$###.##-";12.35,-12.35
  $12.35  $12.35-
Ok
```

"$"**

The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
Example:
PRINT USING "**$#.##";12.35
*$12.35
Ok
```

","

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential format.

Example:

```
PRINT USING "####,.##";1234.5
1,234.50
Ok
PRINT USING "####.##,";1234.5
1234.50,
Ok
```

"^^^^"

Four carats may be placed after the digit position characters to specify exponential format. The four carats allow space for E XX to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or minus sign.

```
Example:
PRINT USING "##.##^^^^";234.56
 2.35E+02
Ok
PRINT USING "#.##^^^^+";-12.34
1.23E+01-
Ok
PRINT USING "#.##^^^^-";-12.34
1.23E+01-
Ok
PRINT USING"+#.##^^^^";12.34,-12.34
+1.23E+01-1.23E+01
Ok
```

"%"

If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. Also, if rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
Example:
PRINT USING "##.##";123.45
%123.45
Ok
PRINT USING ".##";.999
%1.00
Ok
```

If the number of digits specified exceed 24, an "Illegal function call" error will result.

4.1.1.31 READ

Purpose : To read values from a DATA statement and assign them to variables.

Version : Cassette, Disk

Format : READ < list of variables >

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

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

To reread DATA statements from the start, use the RESTORE statement.

Example :

```
10 FOR I = 1 TO 3
20 READ NAM$(I), AGE(I)
30 NEXT
40 DATA JOHN, 42, JOSEPHINE, 24,
   LEO, 21
```

This program reads string and numeric data from the DATA statements in line 40. If a colon follows the name, line 40 should be changed to:

```
40 DATA "JOHN:", 42, "JOSEPHINE:",
   24, "LEO:", 21
```

4.1.1.32 REM

Purpose	:	To allow explanatory remarks to be inserted in a program.
Version	:	Cassette, Disk
Format	:	REM remark

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

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

Remarks may be added to the end of a line by preceding the remark with a apostrophe instead of REM.

Do not use this in a DATA statement as it would be considered as a legal data.

Example :

```
10 '
20 REM CALCULATE DISTANCE TRAVELLED
30 INPUT "AVERAGE VELOCITY" ; V
40 INPUT "TRAVELLING TIME" ; T
50 D = V * T
60 PRINT *DISTANCE COVERED IS" ; D
RUN
AVERAGE VELOCITY? 6
TRAVELLING TIME? 8
DISTANCE COVERTED IS 48
Ok
```

Line 10 shows that (') apostrophe produces the same effect as REM. REM is useful in indicating subroutines in a large program.

4.1.1.33 RENUM

Purpose	:	To renumber program lines.
Version	:	Cassette, Disk
Format	:	RENUM [[<new number>][,[<old number>] [, <increment>]]]

Remarks : <new number> is the first line number to be used in the new sequence. The default is 10.
<old number> is the line in the current program where renumbering is to begin. The default is the first line of the program.
<increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ELSE, ON..GOTO, ON..GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line nnnnn in mmmmm" is printed. The incorrect line number reference (nnnnn) is not changed by RENUM, but line number(mmmmm) may be changed.

NOTE: RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30 only) or to create line numbers greater than 65529. An 'Illegal function call' error will result.

Example : Enter the following program and the respective commands. LIST the program to note the change.

```
10   COLOR 15, 4
12   SCREEN 1
15   LINE (50,50) - (205,141), 8
19   LINE (50,141) - (205,50), 8
23   CIRCLE (128,96), 90, 8
30   PAINT (135,125), 8
40   GOTO 40
```

```
RENUM
```
The entire program is renumbered, starting at 10 with an increment of 10.

```
RENUM 50, 40, 5
```
Renumber the lines from 40 up starting at 50 with an increment of 5.

```
RENUM , , 30
```
Renumber all lines from the lowest line number with an increment of 30. The starting line number is 10.

```
RENUM ,5, 2
```
Renumber from line 5 as 10 with an increment of 2.

```
RENUM 5, ,15
```
Renumber from the first line as 5 with an increment of 15.

```
RENUM 3, 5
```
Starting from line 5, renumber it as 3, with a default increment of 10

4.1.1.34 RESTORE

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

Version : Cassett, Disk

Format : RESTORE [< line number >]

Remarks : After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number > is specified, the next READ statement accesses the first item in the specified DATA statement. If a nonexistent line number is specified, and "Undefined Line number" error will result.

Example :

```
10  DIM X (12)
20  FOR I = 1 TO 3
30  FOR K = 1 TO 4
40  READ X (L)
50  PRINT X (L);
60  NEXT
70  PRINT
80  RESTORE
90  NEXT
100 DATA 1, 2, 3, 4
RUN
1    2    3    4
1    2    3    4
1    2    3    4
Ok
```

The RESTORE statement in line 80 resets the DATA pointer to the beginning. Thus the values in DATA statement are used again.

4.1.1.35 RESUME

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

Version : Cassette, Disk

Format :

```
RESUME
RESUME 0
RESUME NEXT
RESUME <line number>
```

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

RESUME or RESUME 0
  Execution resumes at the statement which caused the error.

RESUME NEXT
  Execution resumes at the statement immediately following the one which caused the error.

RESUME <line number>
  Execution resumes at <line number>.

A RESUME statement that is not in an error trap subroutine causes a "RESUME without error".

Example :

```
10  ON ERROR GOTO 100
20  FOR I = 1 TO 10
30  READ X(I)
40  NEXT
50  DATA 5, 4, 3, 2, 1
60  END
100 IF ERR=4 AND ERL = 30 THEN
    PRINT "LACKING DATA"
110 RESUME 20
RUN
LACKING DATA
Ok
```

Line 100 is the error trapping routine. The RESUME statement on line 110 directs the program flowing back to line 20.

4.1.1.36 RUN

Purpose : To execute a program currently in memory.

Version : Cassette, Disk

Format :
```
RUN [   line number   ]
RUN [   filespec   ] [,R]
```

Remarks : If line number is specified, execution begins on that line. Otherwise, it begins at the lowest line number.

RUN [filespec] loads a file from diskette or cassette into memory and runs it. RUN deletes the current contents of memory, closes all files before loading the program. If the [,R] option is included, all open data files are kept open.

Example : Enter the following program and commands.

```
10 PRINT 10 "x";
20 PRINT 20 "=";
30 M = 10 * 20
40 PRINT M
RUN
10 x 20 = 200
Ok
RUN 30
200
Ok
```

The following example loads the program "TEST" from the disk drive 1 and runs it.

```
  RUN "1: TEST"
```

If the program is stored in tape,
enter the following command.

```
  RUN "TEST"
```

4.1.1.37 STOP

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

Version : Cassette, Disk

Format : STOP

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

```
Break in nnnn    (nnnn is a
                  line number)
```

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

Execution is resumed by issuing a CONT command.

Example :

```
10 FOR I = 1 TO 4
20 INPUT X
30 SUM = SUM + X
40 NEXT
50 STOP
60 PRINT "SUM="; SUM
70 END
RUN
? 4
? 5
? 6
? 7
Break in 50
Ok
CONT
SUM = 22
Ok
```

This example calculates the sum of 4 figures, then stops at line 50; with the printout "Break in 50". Use CONT to resume program execution.

4.1.1.38 SWAP

Purpose : To exchange the value of two variables.

Version : Cassette, Disk

Format : SWAP < variable >,< variable >

Remarks : Any type of variable may be SWAPed (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example :

```
10 A$ = "HEAD" : B$ = "AND" : C$ =
   "TAIL"
20 PRINT A$; B$; C$
30 SWAP A$, C$
40 PRINT A$ + B$ + C$
RUN
HEAD AND TAIL
TAIL AND HEAD
Ok
```

Line 30 renders the content of A$ be changed to "TAIL" while C$ be changed to "HEAD".

4.1.1.39 SWITCH/SWITCH STOP

Purpose : Allow two programs to be stored simultaneously in the RAM.

Version : Disk

Format : SWITCH
SWITCH STOP

Remarks : This caters for the bank program switching of bank 02 and bank 22. This command is only valid as Disk BASIC is run, with the 64K RAM Cartridge installed. Select bank 02 and bank 22 to be switched on.

As the command SWITCH STOP is executed, the program resided in bank 22 will be executed.

Example : Enter the following program which is resided in bank 02:

```
10 PRINT "TESTING SWITCH
   COMMAND"
20 PRINT "NOW BANK 02 IS ON"
```

After the "Ok" prompt, type:

SWITCH $\boxed{\text{ENTER}}$

A clicking noise from the disk drive is heard. The screen is cleared to display the following message:

```
Initializing 2nd bank
Disk version 1.0 by Microsoft
Corp.
Ok.
```

Now enter another program to be resided in bank 22.

```
10 PRINT "NOW BANK 22 IS ON"
```

Then type

```
SWITCH  [ENTER]
RUN [ENTER]
```

You will discover that the first program is executed.

Now try another command:

```
SWITCH STOP
  C
Break
Ok
```

Run the program and you always find the one residing in bank 22 is executed.

4.1.1.40 TRON/TROFF

Purpose : To trace the execution of program statements.

Version : Cassette, Disk

Format : TRON/TROFF

Remarks : As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers are enclosed in square brackets. The trace flag is disabled with the TROFF or NEW command.

Example :

```
10 CLS
20 LOCATE 10, 5
30 PRINT "TEST"
RUN
```

The screen is cleared before the following is displayed:

```
[10]

                      [20] TEST

Ok
```

4.1.2 Functions, except I/O

4.1.2.1 ABS

Purpose : Return the absolute value of an expression.

Version : Cassette, Disk

Format : ABS (X)

Remarks : X may be any numeric expression. The absolute value of a number is always positive or zero.

Example :

```
PRINT ABS (-5 * . 325)
1.625
Ok
```

4.1.2.2 ASC

Purpose : Return the ASCII code for the first character of a string.

Version : Cassette, Disk

Format : ASC (X$)

Remarks : The result of the ASC function is a numerical value that is the ASCII code of the first character of the string X$. If X$ is null, an "Illegal function call" error is returned.

The CHR$ function is the inverse of the ASC function, and it convents the ASCII code to a character.

Refer to Appendix E on "ASCII Characater Code" for details.

Example :

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

This example shows that the ASCII code for "T" is 84.

4.1.2.3 ATN

Purpose : Return the arctangent of a numeric expression in radians.

Version : Cassette, Disk

Format : ATN (X)

Remarks : The expression X may be any numeric type. The evaluation is always performed in double precision. Result lies in the range -Pi/2 to Pi/2.

Convert radian to degree by multiplying a factor of 180/Pi where Pi = 3.141593

Example :

```
10 PI = 3.141593#
20 RAD = ATN (6/8)
30 DEG = RAD * 180/PI
40 PRINT "TAN(";
50 PRINT USING "###.##"; DEG;
60 PRINT ") = 6/8"
RUN
 TAN (36.87) = 6/8
Ok
```

4.1.2.4 BIN$

Purpose : Return a string which represents the binary value of the decimal argument.

Version : Cassette, Disk

Format : BIN$ (n)

Remarks : n is a numeric expression in the range -32768 to 65535. If n is not an integer, its fractional portion is truncated. If n is negative, the two's complement form is used. That is, BIN$(-n) is the same as BIN$ (65536 - n).

Example :

```
PRINT BIN$ (12)
1100
Ok
```

This example shows that the decimal number 12 equals to a binary number of 1100.

4.1.2.5 CDBL

Purpose : Convert a numeric expression to a double precision number.

Version : Cassette, Disk

Format : CDBL

Remarks : X is a numeric expression.
Refer to CINT and CSNG functions for converting numbers to integer and single-precesion respectively.

Example :

```
PRINT CDBL (34/7)
 4.8571428571429
Ok
```

The quotient of 34/7 is given as a double precision number.

4.1.2.6 CHR$

Purpose : To convert an ASCII code to its character equivalent.

Version : Cassette, Disk

Format : CHR$ (I)

Remarks : I lies within the range of 0 to 255. This function returns the one-character string with ASCII code I. CHR$ is usually used to send a special character to the screen or printer.

To convert a character back to its ASCII code, use the ASC function.

Example :

```
PRINT CHR$ (85)
U
Ok
```

This shows that the ASCII code for character "U" is 85.

4.1.2.7 CINT

Purpose : Convert a numeric expression to an integer.

Version : Cassette, Disk

Format : CINT (X)

Remarks : X may be any numeric expression, lying within the range -32768 and 32767.

X is converted to an integer by truncating the fractional portion.

Example :

```
PRINT CINT (45.67)
 45
Ok
```

The fractional portior of 45.67 is truncated to give an integer.

4.1.2.8 COS

Purpose : Return the cosine of a numeric expression in radians.

Version : Cassette, Disk

Format : COS (X)

Remarks : X is the angle whose cosine is going to be calculated. X must be in radians. To convert from degrees to radians, multiply the latter by Pi/180 where Pi = 3.14593.

The calculator of COS(X) is performed in double precision.

Example :

```
PRINT COS (2)
-.4161468365472
Ok
```

4.1.2.9 CSNG

Purpose : Convert a numeric expression to a single precision number.

Version : Cassette, Disk

Format : SNG (X)

Remarks : X is a numeric expression.
See the CINT and CDBL functions for converting numbers to the integer and double-precision value respectively.

Example :

```
10 A = 345.53454663
20 PRINT CSNG (A)
 RUN
  345.535
 Ok
```

Line 20 converts A to a single precision number.

4.1.2.10 CSRLIN

Purpose : Return the vertical coordinate of the cursor.

Version : Cassette, Disk

Format : CSRLIN

Remarks : The CSRLIN variable returns the current line (row) position of the cursor on the active page. The value returned will lie in the range 1 to 25.

Refer to POS function for the column location of the cursor.

Refer to LOCATE statement to see how to set the cursor line.

Example :

```
10 CLS
20 LOCATE 20, 5
30 A = POS (0)
40 B = CSRLIN
```

In this example, the cursor is moved to the 20th row, the 5th column. Then the cursor coordinates are saved in the variables A and B.

4.1.2.11 ERL/ERR

Purpose : Return the error code and line number associated with an error.

Version : Cassette, Disk

Format :
```
ERR
ERL
```

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

If ERL is tested in an IF.....THEN statement, put the line number on the right side of the relational operator, like this:

```
IF ERL =<line number>THEN.....
```

The line number can then be modified as RENUM command is executed.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test whether an error was a direct mode statement, use IF 65535 = ERL THEN..... Otherwise, use IF ERR = <error code> THEN..... or IF ERL = <line number> THEN.....

Example :

```
10  ON ERROR GOTO 100
20  INPUT "NUMBER"; N
30  IF N   100 THEN ERROR 61
40  END
100 IF (ERR = 61) AND (ERL = 30)
    THEN PRINT "TOO
    LARGE": RESUME 20
NUMBER ? 789
TOO LARGE
NUMBER ? 7
Ok
```

In this example, an error trapping routine is set up to check the input number. The RESUME statement on line 100 causes the program to return to line 20 when error 61 occurs in line 30.

4.1.2.12 EXP

Purpsoe : Calculate the exponential function.

Version : Cassette, Disk

Format : EXP (X)

Remarks : X is a numeric expression. X must be = 145.06286085862.

This function returns the mathematical number e raised to the Xth power. e is the base for natural logarithm. If EXP overflows, the "Overflow" error message is printed.

Example :

```
PRINT EXP (2)
 7.38905609893
Ok
```

This example calculates e raised to the second power.

4.1.2.13 FIX

Purpose : Truncate a numeric expression to an integer.

Version : Cassette, Disk

Format : FIX (X)

Remarks : X is a numeric expression.

FIX returns the integer part of X with fraction truncated.

FIX (X) = SGN (X) *INT (ABS (X))

The major difference between the FIX and CSGN is that FIX does not return the next lower number for negative X.

Example :

```
10 PRINT "INT (-34.5) =";
   INT (-34.5)
20 PRINT "FIX (-34.5) =";
   FIX (-34.5)
30 PRINT "SGN (-34.5) *INT
   (ABS (-34.5))
RUN
INT (-34.5) = -35
FIX (-34.5) = -34
SGN (-34.5)* INT
(ABS (-34.5)) = -34
Ok
```

This example shows the difference between INT and FIX functions. FIX(X) is equivalent to SGN(X)* INT (ABS(X)).

4.1.2.14 FRE

Purpose : FRE returns the number of bytes in memory not being used by BASIC.

Versions : Cassette, Disk

Format :

```
FRE (X)
FRE (X$)
```

Remarks : Arguments to FRE are dummy arguments.

FRE (X) returns the number of bytes in memory which can be used for BASIC program, text file, machine language program file, etc. FRE (X $) returns the number of bytes in memory for string space.

Example :

```
10  GOSUB 80
20  DIM A (100) : DIM A$ (100)
30  FOR I = 1 TO 100
40  A(I) = I : A$ (I) = CHR$ (I)
50  NEXT
60  GOSUB 80
70  END
80  PRINT "FRE (0) =" FRE (0),
90  PRINT "FRE(" + CHR$ (34) +
    CHR$ (34) +") = "FRE ("  ")
100 PRINT
110 RETURN
RUN
FRE (0) = 29025
FRE (" ") = 200
FRE (0) = 27887
FRE (" ") = 100
Ok
```

This example shows that A(100), takes up 1138 bytes and A$(100) takes up 100 bytes.

4.1.2.15 HEX$

Purpose : Return a string which represents the hexadecimal value of the decimal argument.

Version : Cassette, Disk

Format : HEX$(n)

Remarks : n is a numeric expression in the range -32768 to 65535. If n is not an integer, its fractional portion is truncated. If n is negative, the two's complement form is used. That is, HEX$(-n) is the same as HEX$(65536-n).

Example :

```
PRINT HEX$(-32768), HEX$(65535)
8000                FFFF
OK
```

This example uses the HEX$ function to figure the hexadecimal representation for the two decimal values which are entered.

4.1.2.16 INKEY$

Purpose : Read a character from the keyboard.

Version : Cassette, Disk

Format : INKEY$

Remarks : Return either a one-character string containing a character read from the keyboard or a null string if no key is pressed. No characters will be echoed and all characters are passed through to the program except for CTRL-C, which terminates the program.

Example :

```
10 PRINT "PRESS ANY KEY TO
   CONTINUE"
20 A$ = INKEY$
30 IF A$ = " " THEN 20
40 CLS
```

This section of a program suspends the program until any key on the keyboard is pressed.

4.1.2.17 INPUT$

Purpose : Return a string of X characters, read from the keyboard.

Version : Cassette, Disk

Format : INPUT$(X)

Remarks : X is the number of characters to be read. No character will be echoed and all characters are passed through except CTRL-C, which terminates the execution of the INPUT$ function.

Example :

```
10 PRINT "INPUT A STRING OF
   TWELVE LETTERS"
20 X$ = INPUT $ (12)
30 IF X $ = " " THEN 20
40 PRINT
50 FOR I = 1 TO 12
60 PRINT TAB(I+10)  MID$ (X$, I,
   1)
70 NEXT
RUN
INPUT A STRING OF TWELVE LETTERS

    S
     P
      E
       C
        T
         R
          A
           V
            I
             D
              E
               O
Ok
```

Line 20 waits for the input via keyboard of 12 characaters.

4.1.2.18 INSTR

Purpose : Search for the first occurrence of string Y$ in X$ and return the position at which the match is found. Optional offset I sets the position for starting the search in X$.

Version : Cassette, Disk

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

Remarks : I must be in the range 0 to 255. X$, Y$ may be string variables, string expressions or string constants. If I   LEN(X$) or if X$ is null or if Y$ cannot be found or if X$ and Y$ are null, INSTR returns 0. If only Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions or string lieterals.

Example : In this example, four characters are read from the keyboard in response to the question.

```
10 A$ = "BEGINNING"
20 B$ = "IN"
30 PRINT INSTR (A$, B$); INSTR (5,
   A$, B$)
RUN
4    7
Ok
```

This example searches for the string "IN" within the string "BEGINNING". When the word "IN" is searched from the first character, it is first found at starting position 4; when the search starts at position 5, it is found at starting position 7.

4.1.2.19 INT

Purpose : Return the largest integer that is less than or equal to X.

Version : Cassette, Disk

Format : INT(X)

Remarks : X is any numeric expression. See the FIX and CINT functions as reference.

Example :

```
PRINT INT (45.6)
45
Ok
```

Since 45.6 = 45 + 0.6, 45 is the largest integer that is less than 45.6.

```
PRINT INT(-45.6)
-46
Ok
```

Since -45.6 = -46 + 0.4, -46 is the largest integer.

4.1.2.20 LEFT$

Purpose : Return a string comprising the leftmost I characters of X$.

Version : Cassette, Disk

Format : LEFT$ (X$, I)

Remarks : X$ is any string expression. I must be in the range 0 to 255. It specifies the number of characters for the result.

If I is greater than total number of characters in X$, the entire string is returned. If I=0, a null string (length = zero) is returned.

Example :

```
PRINT LEFT $ ("RAINDROP", 4)
RAIN
Ok
```

In this example, the LEFT$ function extracts the first four characters from the string "RAINDROP".

4.1.2.21 LEN

Purpose : Return the number of characters in X$.

Version : Cassette, Disk

Format : LEN (X$)

Remarks : X$ is a string expression. Non printing characters and blanks are counted.

Example :

```
PRINT LEN ("LONG ISLAND")
11
Ok
```

There are 11 characters in the string "LONG ISLAND" for spacing is counted as well.

4.1.2.22 LOG

Purpose : Return the natural logarithm of X.

Version : Cassette, Disk

Format : LOG(X)

Remarks : X must be greater than zero. The natural logarithm is the logarithm to the base e.

Example :

```
PRINT LOG(3)
3.1354942159291
Ok
```

The natural logarithm of 3 is 3.1354942159291.

4.1.2.23 LPOS

Purpose : Return the current position of the line printer head within the line printer buffer.

Version : Cassette, Disk

Format : LPOS (X)

Remarks : X is a numeric expression which is a dummy argument.

LPOS function does not necessarily give the physical position of the print head.

Example :

```
IF LPOS(O)   30  THEN LPRINT CHR$
(13)
```

In this example, if the line length is more than 30 characters long then the ENTER character will be sent to the printer so that it skips the next line.

4.1.2.24 MID$

Purpose : Return the requested part of a given string.

Version : Cassette, Disk

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

Remarks : X$ is any string expression. I is an integer expression in the range 1 to 255.
J is an integer expression in the range 0 to 255.

Return a string of length J characters from X$ beginning with the Ith character. If J is omitted or if there are fewer than J characters to the right of the Ith character, all right most characters beginning with the Ith character are returned. If I is larger than total number of characters in X$, MID$ returns a null string.

Example :

```
X$ = "INTERACTION"
Ok
PRINT MID$ (X$, 6, 3)
ACT
Ok
```

The second command prints a string of 3 characters length, starting from the 6th character of X$.

4.1.2.25 OCT$

Purpose : Return a string which represents the octal value of the decimal argument.

Version : Cassette, Disk

Format : OCT$(n)

Remarks : n is a numeric expression in the range -32768 to 65535. If n is negative, the two's complement form is used. That is OCT$ (-n) is the same as OCT$ (65536-n).

Example :

```
PRINT OCT$ (-32768), OCT$ (65535)
100000        177777
Ok
```

The octal value for -32768 is 100000 and that for 65535 is 177777.

4.1.2.26 PEEK

Purpose : Return the byte read from the indicated memory position.

Version : Cassette, Disk

Format : PEEK (I)

Remarks : I is a numeric expression in the range -32768 to 65535.

PEEK is the complementary function to the POKE statement.

Example :

```
10 POKE &H9C40, 5
20 PRINT PEEK(&H9C40); PEEK
   (&O116100);
   PEEK (&B1001110001000000)
RUN
5   5   5
Ok
```

This example shows how a byte being put in a memory location, can be retrieved by PEEK command.
Line 20 reads the byte from memory location 9C40 (in hexadecimal)

4.1.2.27 POS

Purpose : Return the current horizontal cursor position.

Version : Cassette, Disk

Format : POS (I)

Remarks : I is a dummy numeric argument. The left most position is 0. Refer to CSRLIN function for the row location of the cursor.

Example :

```
IF POS(0) > 30  THEN PRINT CHR$
(13)
```

If the cursor is beyond position 30 on the screen when this statement is executed, the cursor will move to the beginning of the next line.

4.1.2.28 RIGHT$

Purpose : Return the right most I characters of string X$.

Version : Casette, Disk

Format : RIGHT $ (X$, I)

Remarks : I must be in the range 0 to 255. It specifies the number of characters for the result.

If I equals to number of characters in X$, the whole string is returned. If I equals to zero, a null string is returned.

Example :

```
 10 X$ = "FOREVER"
 20 PRINT RIGHT$ (X$, 4)
 RUN
 EVER
 Ok
```

The right most four characters of X$ are returned.

4.1.2.29 RND

Purpose : Return a random number between 0 and 1.

Version : Cassette, Disk

Format : RND(X)

Remarks : X is a numeric expression which affects the returned value. The same sequence of random number is generated each time the program is run. To generate a different sequence, use a different value for X each time. If X < 0 , the random generator is reseeded for any given X. X=0 repeats the last number generated. X < 0 generate the next random number in the sequence.

Example :

```
10 FOR I = 1 TO 2
20 PRINT RND (2);
30 NEXT
40 PRINT: PRINT RND (0)
50 FOR I = 1 TO 2
60 PRINT RND(-2);
70 NEXT
RUN
   .59521943994623
   .10658628050158
   .10658628050158
   .94389820420821
   .94389820420821
Ok
```

The first row shows two random numbers, generated using a positive X.

In line 40, RND is called with an argument of zero, so the number generated on the second row is the same as the preceding number.

In line 60, a negative number is used to reseed the random number generator. The random numbers produced after this seeding are in the second row of results.

4.1.2.30 SGN

Purpose	:	Return the sign of X.
Version	:	Cassette, Disk
Format	:	SGN(X)
Remarks	:	X is any numeric expression. For X > 0, it returns 1. For X = 0, it returns 0. For X < 0, it returns -1.
Example	:	ON SGN(X) + 2 GOTO 100, 200, 300

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

This statement directs the program branch to 100 if X is negative 200 if X is zero and 300 if X is positive.

4.1.2.31 SIN

Purpose : Return the sine of X in radians.

Version : Cassette, Disk

Format : SIN(X)

Remarks : X is an angle in radians. To convert degrees to radians, multiply by Pi/180, where Pi=3.141593. SIN(X) is calculated to double precision.

Example :

```
PRINT SIN(0)
0
Ok
```

4.1.2.32 SPACE$

Purpose : Return the string of spaces of length X.

Version : Cassette, Disk

Format : SPACE$ (X)

Remarks : The expression X discards the fractional portion and must be in the range 0 to 255.
Refer also to SPC function.

Example :

```
10 PRINT "_"
20 FOR I = 1 TO 5
30 X$ = SPACE$ (I)
40 PRINT X$: "L"
50 NEXT
RUN

_
 L
  L
   L
    L
     L
Ok
```

This example uses the SPACE$ function to print the character "L" on a line preceded by I spaces. Notice BASIC puts a space in front of character "L".

4.1.2.33 SPC

Purpose : Print I blanks on the screen.

Version : Cassette, Disk

Format : SPC (I)

Remarks : I must be in the range 0 to 255. SPC may only be used with PRINT and LPRINT statements. The SPC function has implied semicolon after it.

Example :

```
PRINT "MAGNETIC" SPC(5) "FIELD"
 MAGNETIC     FIELD
Ok
```

Notice there are five spaces between "MAGNETIC" and "FIELD".

4.1.2.34 SQR

Purpose : Return the square root.

Version : Cassette, Disk

Format : SQR(X)

Remarks : X must be greater than or equal to zero. This function returns the square root of X.

Example :

```
PRINT SQR (25)
5
Ok
```

This example calculates the square root of 25.

4.1.2.35 STR$

Purpose : Return a string representation of the value of X.

Version : Cassette, Disk

Format : STR$(X)

Remarks : X is any numeric expression. If X is positive, the string returned by STR$ contains a leading blank. The VAL function is complementary to STR$.

Example :

```
PRINT STR$ (8*7); LEN (STR$ (8*7))
 56    3
Ok
```

Eight times seven gives fifty-six. STR$ then converts the digits in the number to a string. Notice that there is a leading space in the returned string.

4.1.2.36 STRING$

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

Version : Cassette, Disk

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

Remarks : I, J are in the range 0 to 255. X$ is any string expression.

Example :
```
10 X$ = "FLUTE"
20 Y$ = STRING $ (5, 42)
30 PRINT Y$ + "PU" + STRING $ (2,
   X$) + Y$
RUN
***** PUFF *****
Ok
```

On line 20, a string consisting of five asterisks is assigned to Y$. On line 30, STRING $ (2, X$) extracts the first character from X$ and repeats the latter twice to form another string.

4.1.2.37 TAB

Purpose : Space to position I on the console.

Version : Cassette, Disk

Format : TAB(I)

Remarks : I is a numeric expression in the range 0 to 255.

If the current print position is already beyond space I, TAB does nothing. Space 0 is the left most position, and the right most position is the width minus one. TAB may only be used with PRINT and LPRINT statements.

Example :

```
10 A$ = "NEW": B= "GENERATION"
20 PRINT A$ TAB (10) B$
30 PRINT A$ SPC (7) B$
RUN
NEW       GENERATION
NEW       GENERATION
Ok
```

Line 20 commands printing of B$ at the 10th column. Notice the same effect is produced by using SPC function on line 30.

4.1.2.38 TAN

Purpose : Return the tangent of X in radians.

Version : Cassette, Disk

Format : TAN(X)

Remarks : X is the angle in radians. To convert degrees to radians, multiply by Pi/180, where Pi = 3.141593. TAN(X) is calculated to double precision. If TAN overflows, an "Overflow" error will occur.

Example :

```
PRINT TAN(0)
0
Ok
```

4.1.2.39 USR

Purpose : Call the user's assembly language subroutine with the argument X.

Version : Cassette, Disk

Format : USR [digit] (X)

Remarks : digit is in the range 0 to 9 and corresponds to the digit supplied with the DEFUSR statement for that routine. If digit is omitted, USR0 is assumed. X is any numeric expression of the argument to the machine language subroutine.

Example :

```
10 DEF USR0 = &HF000
20 C = USR0 (B/2)
30 D = USR0 (B/3)
```

The function USR0 is defined on line 10. Line 20 calls the functions USR0 with the argument B/2 while line 30 calls USR0 again, with the argument B/3.

4.1.2.40 VAL

Purpose : Return the numeric value of a string.

Version : Cassette, Disk

Format : VAL (X$)

Remarks : X$ is a string expression.

The VAL function returns the numeric value of a string, also strips leading blanks, tabs and linefeeds from the argument string.

If the first character of X$ is not numeric, then VAL(X$) will return 0.

Refer to STR$ function for numeric to string conversion.

Example

```
PRINT VAL ("420 BOAR LANE")
420
Ok
```

The VAL function returns only the numeric value (420) from a string Both the leading space and the trailing characters are stripped.

4.1.2.41 VARPTR

Purpose : Return the address in memory of the variable or file control block.

Version : Cassette, Disk

Format : VARPTR (< variable name >)
VARPTR (# < filenumber >)

Remarks : < variable name > is the name of a numeric or string variable or array element. A value must be assigned to variable name prior to execution of VARPTR. Otherwise an "Illegal function call" error results.

< filenumber > is the number under which the file was opened.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to a machine language subroutine. The address returned will be an integer in the range -32768 to 32767. If the negative address is returned, add it to 65536 to obtain the actual address. If < filenumber > is specified, VARPTR returns the starting address of the file control block.

A function call of the form VARPTR (A (0)) is usually specified when passing an array, so that the lowest addressed element of the array is returned. All simple variables should be assigned before calling VARPTR for an array because the address of the array changes whenever a new simple variable is assigned.

Example :

```
10 A$ = "SUPERLATIVE"
20 B = VARPTR(A$)
30 PRINT HEX$ (B)
RUN
8033
Ok
```

This example uses VARPTR to get the data from a variable. In line 20, B gets the address of the data. Then it is converted to a hexadecimal figure.

1st line of 4.2 **DEVICE SPECIFIC STATEMENTS AND FUNCTIONS**

4.2.1 Statements

4.2.1.1 BEEP

Purpose : To generate a beep sound.

Version : Cassette, Disk

Format : BEEP

Remarks : Exactly the same with the command PRINT CHR$(7).

Example :

```
10 FOR T = 1 TO 10
20 BEEP
30 NEXT
40 PRINT "* * * * *"
50 FOR T = 1 TO 10
60 PRINT CHR$(7)
70 NEXT
```

It beeps ten times before and after the printout of a string of five asterisks.

Both lines 20 and 60 produce beeping sound.

4.2.1.2 BLOAD

Purpose : To load a machine language program from the specified device.

Version : Cassette, Disk

Format : BLOAD " < device descriptor > [< filename >]" [, R], [, < offset >]

Remarks : The < device descriptor > can be one of the followings : CAS: , 1: or 2:.

< filename > : Refer to section 3.13.1.2.2.

If "R" option is specified, after the loading, program begins execution automaticaly from the address which is specified at BSAVE.

The loaded machine language program will be stored at the memory location which is specified at BSAVE. If < offset > is specified, all addresses which are specified at BSAVE are offset by that value.

If the < filename > is omitted, the next machine language program file encoutered is loaded.

Example :

```
BLOAD "1 : SVFRMT", R
```

The file named "SVFRMT" is loaded from disk on drive 1 and is run.

4.2.1.3 BSAVE

Purpose : To save a memory image at the specified memory location to the device.

Version : Cassette, Disk

Format : BSAVE " < device descriptor [< filename >]", < top adrs > , < end adrs > [, < execution adrs >]

Remarks : The < device descriptor > can be one of the followings: CAS: , 1: or 2:. This may be omitted if the device is cassette.

< top adrs > and < end adrs > are the top address and the end address of the area to be saved.

If < execution adrs > is omitted, < top adrs> is regarded as < execution adrs>.

Bsave is useful for saving machine language program.

Exmaple :

```
BSAVE "CAS: TEST", &HA000, &HAFFF
```

The file "TEST" is saved on cassette starting at address &H A000 and ending at &HAFFF.

4.2.1.4 CIRCLE

Purpose : To draw an ellipse with a center and radius as indicated by the first of its arguments.

Version : Cassette, Disk

Format :
```
CIRCLE < coordinate specifier > ,
< radius>[, < color > ]
[, < start angle > ]
[, < end angle> ]
[, < aspect ratio > ]
```

Remarks : < coordinate specifier > specifies the coordinate of the center of the circle on the screen. For the detail of <coordinate specifier>, see the description at PUT SPRITE statement.

The < color > defaults to foreground color.

The < start angle > and < end angle > parameters are radian arguments between 0 and 2*Pi which you specify where drawing of the ellipse will begin and end. If the start or end angle is negative, the ellipse will be connected to the center point with a line, and the angles will be treated as if they were positive. Note that this is different from adding 2*Pi.

The < aspect ratio > is the height/width ratio of the ellipse. The default is 1, assuming a monitor screen ratio of 4/3. If the < aspect ratio > is less than 1, the radius specifies y pixels. If the ratio is larger than 1, the radius specifies x pixels.

Example :

```
10 SCREEN 1
20 CIRCLE (128, 96), 80, 15
30 GOTO 30
```

A white circle centered at (128, 96) with a radius of 80 is displayed.

Now change line 20 to:

```
   20 CIRCLE (128, 96), 80, 15, 0,
   3.14
```

Run the program. Only the upper half circle is drawn.

Change line 20 to:

```
   20 CIRCLE (128, 96), 80, 15,,,2
```

Then,

```
   20 CIRCLE (128, 96), 80,
   15,,,.5
```

These will draw ellipses. The three commas after the number 15 are necessary to inform the computer that the starting and ending points of the shape to be drawn are not specified. It assumes the complete shape to be drawn.

4.2.1.5 CLOAD

Purpose	: To load a BASIC ogram file from the cassette motor.
Version	: Cassette
Format	: CLOAD [" <filename> "]
Remarks	: <filename> is a string of characters, six being the maximum. CLOAD closes all open files and deletes the current program from memory. If the <filename> is omitted, the next program file encountered on the tape is loaded. For all cassette read operations, baud rate is determined automatically.
Example	: CLOAD "INTRO" The file named "INTRO" is read from the cassette onto the computer.

4.2.1.6 CLOAD?

Purpose : To verify a BASIC program on cassette motor with one in memory.

Version : Cassette

Format : CLOAD? [" <filename> "]

Remarks : <filename> is a string of characters, six being the maximum.

If the program loaded is different from the one in memory the message "verify error" is displayed.

Example : CLOAD? [" <filename> "]
To verify a BASIC program on cassette motor with one currently in memory.

4.2.1.7 CLOSE

Purpose : To close the channel and release the buffer associated with it.

Version : Cassette, Disk

Format : CLOSE [[#] <filenumber> [, < filenumber>]]

Remarks : <filenumber> is the number used on the OPEN statement. As CLOSE is executed, any association between a file and device stops. Subsequent I/O operations specifying that file number will be invalid. The file or device should be OPEN again.

A CLOSE with no file number specified causes all devices and files that have been opened to be closed.

Example :

```
10 OPEN "1 : DEMO" FOR OUTPUT AS #
   1
20 FOR I = 0 TO 50
30 PRINT # 1, I
40 NEXT I
50 CLOSE # 1
```

On line 50, the file is closed after data has been written to it.

4.2.1.8 CLS

Purpose : To clear the screen.

Version : Cassette, Disk

Format : CLS

Remarks : Erase the current active screen page. The CLS statement returns the cursor to home position in the upper left-hand corner of the screen.

The SCREEN statement will force a screen clear if the resultant screen mode created is different from the mode currently in force. So is WIDTH statements.

The screen may also be cleared by depressing the CLS or CTRL and L keys simultaneously. Or else use PRINT CHR$ (12).

4.2.1.9 COLOR

Purpose : Set the colors for the foreground, background and border screen.

Version : Cassette, Disk

Format : COLOR [< foreground color >]
[, < background color >]
[, < border color >]

Remarks : Each character on the screen is composed of two parts: foreground and background. The foreground is the character itself. The background is the "box" around the character. Most TV or monitors have an overscan area outside the area for characters. This is known as border screen.

The arguments lie in the range 0 to 15. Default is 15,4,5. The sixteen colors are:

0	transparent
1	black
2	medium green
3	light green
4	dark blue
5	light blue
6	dark red
7	cyan
8	medium red
9	light red
10	dark yellow
11	light yellow
12	dark green
13	magenta
14	gray
15	white

Example :

```
10 SCREEN 2
20 FOR I = 0 TO 7
30 CLS
40 COLOR I, I + 8
50 FOR T = 1 TO 5
60 LOCATE 10, 40 : PRINT "COLOR"
70 LOCATE 5, 80 : PRINT I "," I +
   8
80 NEXT T, I
90 COLOR 15, 4, 5
```

Both the background and character colors change as the above program is executed. The respective color number are printed on the screen.

Line 40 sets the foreground color, i.e., color of text, to be I and the background to be I + 8.
Line 90 sets the foreground color as white(15), while the background color as dark blue(4) and the border color as light blue (5).

4.2.1.10 CSAVE

Purpose : To save a BASIC program file to the cassette tape.

Version : Cassette

Format : CSAVE " <filename> "

Remarks : <filename> is the name for the program to be saved on cassette. The maximum number of characters is six.

BASIC saves the file in a compressed binary(tokenized) format. ASCII files take up more space, but some types of access require that file to be in ASCII format. Programs saved in ASCII may be read as BASIC data files and text files. In that case, use the SAVE command.

Example :

```
CSAVE "DEMO"
```

The program currently in memory is named "DEMO" and is saved on cassette.

4.2.1.11 DRAW

Purpose	: To draw figure according to the graphic macro language.
Version	: Cassette, Disk
Format	: DRAW < string expression >

Remarks : The graphic macro language commands are contained in the string expression string. The string defines an object, which is drawn when BASIC executes the DRAW statement. During execution, BASIC examines the value of string and interprets single letter commands from the contents of the string. These commands are detailed below.

The following movement commands begin movement from the last point referenced. After each command, the last point referenced is the last point the command draws.

U	**n**	Move up
D	**n**	Move down
L	**n**	Move left
R	**n**	Move right
E	**n**	Move diagonally up and right
F	**n**	Move diagonally down and right
G	**n**	Move diagonally down and left
H	**n**	Move diagonally up and left

n in each of the preceding commands indicates the distance to move. The number of points moved is n times the scaling factor (set by the S command).

M <x,y >	Move absolute or relative.dis If x has a plus sign (+) or a minus sign (-) in front of it, it is relative. Otherwise, it is absolute.

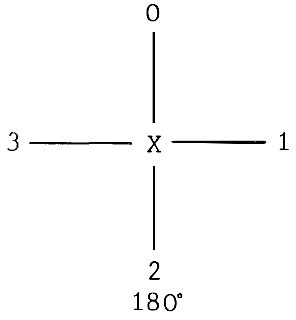
The aspect ratio of the screen is 1. So 8 horizontal points are equal in length to 8 verrtical points.

The following two prefix commands may precede any of the above movement commands.

B	Move, but plot no points.
N	Move, but return to the original position when finished.

The following commands are also available:

A <n >	Turn an angle. n may be 0 or 2; 0 for 0 and 2 for 180 .



C <n>	Set color n. n may range 0 to 15.
S <n>	Set scale factor. n may range from 0 to 255. n divided by 4 is the scale factor. For example, if n=1, then the scale factor is 1/4. The scale factor multiplied by the distance given with the U,D,L,R,E,F, G,H relative to M command gives the actual distance moved. The default value is 0, which means no-scaling i.e., same as S4.
X <string variable>	Execute a substring. This allows you to execute a second string from within a string.

In all of these commands, the n, x, or y argument can be a constant like 123 or it can be expressed as" = variable ;" where variable is the name of a numeric variable. The semicolon (;) is required when you use a variable in this way, or in the X command. Otherwise, a semicolon is optional between commands. Spaces are ignored in string. For example, you could use variables in a move command this way:

```
M + = X1;, = X2;
```

The X command can be a very useful part of DRAW, because you can define a part of an object separated from the entire object and also can use X to draw a string of commands more than 255 characters long.

Example :

```
10 COLOR, 1, 1 : SCREEN 1
20 DRAW "C10 BM 100, 70 E15 R30
   G15 L30 D30 R30 U30"
30 DRAW "S4C8 BM 130, 100 E15
   U30"
40 GOTO 40
```

A cube is drawn in two brushes, one in yellow and the other in red, starting at (100, 70) and (130, 100) respectively. Scale factor S4 needs not be specified since the default is 4. (Compare the effect of line 20 and line 30.) Replace lines 20 and 30 by the following line.

```
20 DRAW "C8 BM 100, 70 E15 R30
   D30 G15 L30 U30 R30 NE15 D30"
```

Remember to delete line 30 before executing the modified program.

```
10 COLOR 7, 1 : SCREEN 1
20 DRAW "S4BM 100, 100 E50 F50
   G50 H50"
30 DRAW "S2BM + 25, 25 U50 R50
   D50 L50"
40 GOTO 40
```

A smaller square as drawn per line 30 is enclosed by a larger square as drawn per line 20.

```
10 SCREEN 1 : COLOR 7, 1
20 DRAW "BM 100, 50 F30 L60
   E30"
30 X=-40 : Y=40
40 DRAW "BM + = X;, = Y; R80
   F30 L140 E30"
50 GOTO 50
```

A triangle and a trapezium, seperated from each other are displayed. Line 40 causes the trapezium to be drawn at 40 units left and 40 units below the last referenced point i.e., (100, 50), which needs to be traced back from line 20.

```
10 SCREEN 1
20 T$ = "L20 D20 R20;"
30 DRAW "BM 100, 100 A0 x T$;"
40 GOTO 40
```

In this example, a "[" shape is displayed.
Line 30 reads like this : starting at point (100,100) at an angle of zero degree to the vertical, draw the substring (T$). The effect of angle setting can be seen if you change "A0" on line 30 to "A2".

4.2.1.12 GET

Purpose : To read a record from a random disk file into a random buffer.

Version : Disk

Format : GET[#]<filenumber> [, <record number>]

Remarks : <filenumber> is the number under which the file was OPENed. If < record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

After a GET statement, INPUT # and LINE INPUT # may be done to read characters from the random file buffer.

Example :

```
10 INPUT "NAME:" ; NAME$
20 INPUT "OCCUPATION:"; JOB$
30 OPEN "1:RECORD" AS#1
40 FIELD #1, 20 AS NAME$, 20 AS JOB$
50 LSET N$ = NAME$
60 LSET J$ = JOB$
70 PUT #1, 18
80 GET #1, 18
90 CLOSE #1
```

A random file named "RECORD" under the filenumber (#1) is created per lines 30 to 70. Line 80 moves the desired record (record # 18) into the random buffer.

4.2.1.13 GET (graphics)

Purpose : To read points from an area of the screen.

Version : Cassette, Disk

Fromat : GET <array name>
GET (x1, y1) - (x2, y2), <array name>

Remarks :

(x1, y1), (x2, y2) are the coordinates of the opposite corners of the screen area.

GET reads the colors of the points in the specified screen area into the array. The rectangular area of the screen has its opposite corners as points (x1, y1) and (x2, y2).

Example : The array only holds the image in the specified area without concern as to precision but it must be in numeric form.

```
10  SCREEN 1
20  DEFINT C
30  CIRCLE (128, 96), 8
40  LINE (128, 164) - (128, 118)
50  DRAW "BM 128, 104 G15"
60  DRAW "BM 128, 104 F15"
70  DRAW "BM 128, 118 G15"
80  DRAW "BM 128, 118 F15"
90  DIM C (30, 35)
100 GET (118, 88) - (138, 128), C
110 PUT (20, 16), C, PSET
```

Line 30 through 80 represent the figure drawn. Line 90 creates a rectangular array large enough to hold it. Line 100 GETs the information that follows the word GET and places it in container "C". This line will take the two sets of points that are specified, which had they been drawn would have created a rectangle and place this picture in "C". Line 110 simply PUTs the contents of "C" at location (20, 16).

4.2.1.14 INPUT

Purpose : To read data items from the specified channel and assign them to program variables.

Version : Cassette, Disk

Format : INPUT # <filenumber> , <variable list>

Remarks : <filenumber> is the number used when the file was opened for input. <variable list> is the name of a variable that will have an item in the file assigned to it. It may be a string or numeric variable, or an array element. The type of data in the file must match the type specified by the <variable list> . Unlike the INPUT statement, no question mark is printed with INPUT # statement.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, enters and line feeds are ignored. The first character encountered which is not a space, enter or line feed is assumed to be start of a number. The number terminates on a space, enter, line feed or comma.

Also, if BASIC is scanning the data for a string item, leading spaces, enters and line feeds are ignored. The first character encountered is assumed to be the start of a string item. If this first character is a double-quotation mark ("), the string item will consist of all characters characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character.

If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, enter, line feed or after 255 characters have been read. If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

Example :

```
10 OPEN "1 = DEMO" FOR OUTPUT AS # 1
20 A=10 : B=20 : C=30
30 PRINT # 1, A; B; C
40 CLOSE # 1
50 OPEN "1 : DEMO" FOR INPUT AS # 1
60 INPUT # 1, A, B, C
70 CLOSE # 1
```

This program will save the numbers 10, 20 and 30 on the disk then read them.

On line 50 the computer is instructed to reopen the file. Notice that the filenumber is # 1.

Line 60 causes the computer to read the information back into the computer.

4.2.1.15 INPUT$

Purpose : To return a string of n characters, read from the keyboard or from a specified file.

Version : Cassette, Disk

Format : INPUT$ (<n> , [#] <filenumber>)

Remarks : <n> is the number of characters to be read from the file.

<filenumber> is the number which the file was OPENed.

If the keyboard is used for input, no characters will be displayed on the screen. All characters including control characters are passed through except CTRL + STOP. The latter is used to interrupt the execution. Response to INPUT$ from the keyboard need not press ENTER.

Example :

```
10 PRINT "IS THE STATEMENT
   CORRECT?"
20 Z$ = INPUT$(1)
30 IF Z$ = "Y" OR Z$ = "y" THEN
   PRINT "ARE YOU SURE?"
40 IF Z$ = "N" OR Z$ = "n" THEN
   PRINT "THAT'S CORRECT!" ELSE
   PRINT "COME ON, MAKE UP YOUR
   MIND!"
```

Line 20 collects one single character input via the keyboard.

4.2.1.16 INTERVAL ON/OFF/STOP

Purpose	:	To activate/deactivate trapping of time interval in a BASIC program.
Version	:	Cassette, Disk
Format	:	INTERVAL ON/OFF/STOP
Remarks	:	An INTERVAL ON statement must be executed to activate trapping of time interval. After INTERVAL ON statement, if a line number is specified in the ON INTERVAL GOSUB statement then every time BASIC starts a new statement it will check the time interval and accordingly perform a GOSUB to the line number specified in the ON INTERVAL GOSUB statement. If an INTERVAL OFF statement has been executed, no trapping takes place and the event is not remembered even if it does take place. If an INTERVAL STOP statement has been executed, no trapping will take place, but if the timer interrupt occurs, this is remembered so an immediate trap will take place when INTERVAL ON is executed.
Example	:	Refer to ON INTERVAL GOSUB.

4.2.1.17 KEY

Purpose : To set each function key to automatically type any sequence of characters.

Version : Cassette, Disk

Format : KEY <function key #> , <string expression>

Remarks : <function key #> is the key number - an unsigned integer in the range 1 to 10.

<string expression> is the key assignment text - any valid string expression within 15 characters. If the string is longer than 15 characters, only the first 15 characters are assigned.

The defined string expression is input as a BASIC command, when the assigned function key is depressed.

To disable the function key as a soft key, assign a null string to the latter.

Example : Assign the string "PRINT TIME$" and ENTER to function key #1. Using the following command:

```
KEY 1, "PRINT TIME$" + CHR$(13)
```

Another way to specify a function key, without including the ENTER command may be:

```
A$ = "NUMBER"
KEY 2, A$
```

To disable a function key, use the following command:

```
KEY 1, "  "
```

4.2.1.18 KEY LIST

Purpose : To list the contents of all function keys.

Version : Cassette, Disk

Format : KEY LIST

Remarks : This command lists all ten function key values on the screen. All 15 characters are assigned. Position in the list reflects the key assignments. Note that control characters assigned to a function key is converted to spaces.

Example :

```
KEY LIST
color             auto
goto              list
run               color 15, 4, 5
cload"            cont
list              run
Ok
```

Initially, the function keys are assigned the above values.

4.2.1.19 KEY ON/OFF/STOP

Purpose :	To activate/deactivate trapping of the specified function key in a BASIC program.
Version :	Cassette, Disk
Format :	KEY (< function key # >) ON/OFF/STOP
Remarks :	< function key # > is a numeric expression in the range 1 to 14. A KEY(n)ON statement must be executed to activate trapping of function key. After KEY(n)ON statement, if a line number is specified in the ON KEY GOSUB statement then every time BASIC starts a new statement it will check to see if the specified key was pressed. If so it will perform a GOSUB to the line number specified in the ON KEY GOSUB statement. If a KEY(n)OFF statement has been executed, no trapping takes place and the event is not embered even if it does take place. If a KEY(n)STOP statement has been executed, no trapping will take place, but if the specified key is pressed this is remembered so an immediate trap will take place when KEY(n)ON is executed. KEY(n)ON has no effect on the assigned text of the function key displayed at the bottom of the screen.
Example :	Refer to ON KEY GOSUB.

4.2.1.20 LINE

Purpose : To draw line connecting the two specified coordinates. For the detail of the coordinate specifier , see description at PUT SPRITE statement.

Version : Cassette, Disk

Format : LINE [<coordinate specifier>] - <coordinate specifier> [, <color>] [, B/BF]

Remarks : If the starting pair of coordinates are omitted, a line will be drawn from the last reference point to the position specified by the second pair of coordinates. The default is (0,0). The second pair of coordinates can be written in relative form, by adding a specified offset to the coordinates of the first point. For example, LINE (100, 100) - STEP (20, -20) produces the same effect as LINE (100, 100) - (120, 80).

"B" renders a rectangle to be drawn, with line specified by the pair of coordinates as its diagonal.

"BF" signifies the box thus drawn as in the "B" mode to be painted in the color same as the border of the box.

Example :

```
10 SCREEN 1
20 LINE (72, 72) - (200, 168), 15,
   B
30 LINE (72, 72) - (136, 36)
40 LINE - (200, 72)
50 LINE - (72, 72)
60 LINE (120, 108) - (152, 168),,
   BF
70 GOTO 70
```

Line 40 commands a line to be drawn from the last referenced point (136, 36) to (200, 72). Similar for line 50.

4.2.1.21 LINE INPUT#

Purpose :	To read an entire line (up to 254 characters), without delimiters, from a sequential file to a string variable.
Version :	Cassette, Disk
Format :	LINE INPUT#<filenumber> , <string variable>
Remarks :	<filenumber> is the number which the file was OPENed.

<string variable> is the name of a string variable to which the line will be assigned.

LINE INPUT# reads all charactes in the sequential file up to an enter. It then skips over the enter/line feed sequence, and the next LINE INPUT# reads all characters up to the next enter. If a line feed/enter sequence is encountered, it is preserved. That is, the line feed/enter characters are returned as part of the string.

LINE INPUT# is especially useful if each line of a file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

Example :

```
10 OPEN "1: DEMO" FOR OUTPUT AS # 1
20 A$ = "THIS IS A DEMO"
30 B$ = "APPENDIX"
40 PRINT # 1, A$, B$
50 CLOSE # 1
60 OPEN "1: DEMO" FOR INPUT AS # 1
70 LINE INPUT # 1, A$
80 CLOSE # 1
```

This program writes the message contained on lines 20 and 30 on the disk, then reads it back. The command LINE INPUT# reads an entire line up to 254 characters from a sequential file to a string variable.

4.2.1.22 LOAD

Purpose	:	To load a BASIC program from the device.
Version	:	Cassette, Disk
Format	:	LOAD " [<device descriptor>] [<filename>]" [,R]
Remarks	:	<device descriptor> : For cassette, this may be "CAS:" or just omit this specifier. For disk, this may be "1:" or "2:", depending on the disk drive in use.

<filename> Refer to section 3.13.1.2

LOAD closes all open files and deletes the current program from memory. However, with the "R" options all data files remain OPEN and execute the loaded program.

If the < filename > is omitted, the next program, which should be an ASCII file, encounted on the tape is loaded.

If the "R" option is included, the program is run after it is loaded. In this case, all open data files are kept open. Thus LOAD with the "R" option may be used to chain several programs or segments of the same program. Information may be passed between the programs using data files. This is equivalent to RUN.

Example : LOAD "NOMIS"
Load the program "NOMIS" but does not run it.

LOAD "NOMIS", R
Load and run the program "NOMIS" residing on cassette.

4.2.1.23 LOCATE

Purpose : To locate character position for PRINT.

Version : Cassette, Disk

Format : LOCATE [<x>] [, <y>] [, <cursor display switch>]

Remarks : <x> is a numeric expression in the range 1 to 40 or 1 to 80, depending upon screen width.

<y> is a numeric expression in the range 1 to 25. It indicates the screen line number where you want to place the cursor.

<cursor display switch> is a value indicating whether the cursor is visible or not. A zero (0) indicates off, one (1) indicates on. Valid in the text mode.

Example :

```
10 CLS
20 LOCATE 5,5
30 PRINT "BALANCE SHEET"
40 LOCATE 5, 10, 1
50 PRINT "AMOUNT"
60 GOTO 60
```

This example prints out "BALANCE SHEET" and "AMOUNT" in two seperate lines.

Line 20 locates the PRINT position to the fifth row and the fifth column. Line 40 locates the PRINT position to the tenth row and the fifth column. The cursor prompt is displayed on the eleventh row and the first column.

4.2.1.24 LSET AND RSET

Purpose : To move data from memory to a random file buffer in preparation for a PUT statement.

Version : Disk

Format : LSET < string variable > = < string expression >
RSET < string variable > = < string expression >

Remarks : If < string expression > requires fewer bytes than were FIELDed to < string variable > , LSET left-justified the string. Spaces are used to pad the extra positions. If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See the MKI$, MKS$, MKD$ functions.

Example :

```
50 LSET A$ = MKS$(AMT)
60 LSET D$ = DESC$
```

The LSET commands in lines 50 and 60 move the data from the MKS$(AMT) and DES$ and place it into the string variables, A$ and D$ which are in the random buffer.

LSET or RSET may also be used within a non-fielded string variable to left-justify or right -justify a string in a given field.

```
110 A$ = SPACE$(20)
120 REST A$ = N$
```

The above two lines right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

4.2.1.25 MAXFILES

Purpose : To specify the maximum number of files opened at a time.

Version : Cassette, Disk

Format : MAXFILES = < expression >

Remarks : < expression > can be in the range of 0 to 15. When "MAXFILES=0" is executed, only SAVE and LOAD can be performed.

The default value asssigned is 1.

Example :

```
10 MAXFILES = 3
20 OPEN "CAS:INDEX" FOR INPUT AS
   #1
30 OPEN "CRT:CHAP 1" As # 2
40 OPEN "KYBD:CHAP 2" As # 3
.
.
.
.
.
```

Line 10 specifies the maximum number of files opened be 3.

4.2.1.26 MERGE

Purpose : To merge the lines from an ASCII program file into the program currently in memory.

Version : Cassette, Disk

Format : MERGE "<device descriptor> [< filename>]"

Remarks : <device descriptor> : This may be "CAS:", "1:" or "2:". If this is omitted, the device descriptor is cassette.

<filename> : Refer to section 3.13.1.2.2.

If any lines in the file being merged have the same line number as lines in the program in memory, the lines from the file will replace the corresponding lines in memory.

After the MERGE command, the merged program resides in memory, and BASIC returns to command level.

If the < filename > is omitted, the next program file, which should be ASCII file, encountered on the tape is MERGEd.

Example :

```
MERGE "1: TEST"
```

This command merges the file named "TEST" on diskette in drive 1 with the program in memory.

The program "TEST" should be stored as an ASCII file. The program line numbers are merged with the line numbers of the program that resided in memory before the "merge" was performed.

4.2.1.27 MOTOR ON/OFF

Purpose	:	To change the status of cassette motor switch.
Version	:	Cassette, Disk
Format	:	MOTOR ON/OFF
Remarks	:	When no argument is given, flips the motor switch. Otherwise, enables/ disables motor of cassette.

4.2.1.28 ON INTERVAL GOSUB

Purpose : To set up a line number for BASIC to trap to at defined time interval.

Version : Cassette, Disk

Format : ON INTERVAL = <time interval> GOSUB <line number>

Remarks : Generate a timer interrupt at every <time interval> /60 seconds.

When the trap occurs an automatic INTERVAL STOP is executed so receive traps can never take place. The RETURN from the trap routine will automatically do an INTERVAL ON unless an explicit INTERVAL OFF has been performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place this automatically disables all traps (including ERROR, STRIG, STOP, SPRITE, INTERVAL and KEY).

Example :

```
10   ON INTERVAL = 60 GOSUB 100
20   INTERVAL OFF
30   FOR I = 1 TO 100
40   PRINT I;
50   NEXT
60   INTERVAL ON
70   GOTO 70
100  BEEP : RETURN
```

After printing integers from 1 to 100, the computer beeps every second.

Line 10 directs the program flow to
line 100 every second. That is a
beep is sound every second.
However, the INTERVAL OFF command on
line 20 disables this trapping.
After printing 100 integers, the
INTERVAL ON command is executed.
Beep sound is heard.

4.2.1.29 ON KEY GOSUB

Purpose	:	To set up line numbers for BASIC to trap to when the respective function key is pressed.
Version	:	Cassette, Disk
Format	:	ON KEY GOSUB <list of line numbers>
Remarks	:	If the first line number which is not 0 of an interrupt handling routine is assigned in an ON KEY GOSUB statement, a check will be performed to see if the assigned key is depressed each time BASIC executes a statement. If the key is depressed, BASIC will branch to the routine with the assigned line number.

When a trap occurs, an automatic KEY(n) STOP is executed so receive traps can never take place. The RETURN from the trap routine will automatically do a KEY(n) ON unless an explicit KEY(n) OFF has been performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place this automatically disables all trapping (including ERROR, STRIG, STOP, SPRITE, INTERVAL and KEY).

Example :

```
10   ON KEY GOSUB 50, 90, 80
20   KEY (1) ON : KEY (2) ON : KEY
     (3) ON
30   CLS
40   C = 1
50   COLOR C
60   PRINT "*";
70   GOTO 60
80   BEEP
90   C = C + 1
100  IF C = 16 THEN C = 1
110  RETURN 50
```

As the program is executed, press F1, color of "*" remains the same as the key has not been pressed. Press F2 will change color of "*". Press F3, color will be changed and a beep is heard.

Change line 20 to 20 KEY(1) OFF : KEY(2) OFF: KEY(3) OFF

Notice that the printout color does not change even F2 or F3 is pressed.

4.2.1.30 ON SPRITE GOSUB

Purpose	:	To set up a line number for BASIC to trap to when the sprites coincide.
Version	:	Cassette, Disk
Format	:	ON SPRITE GOSUB <line number>
Remarks	:	When the trap occurs an automatic SPRITE STOP is executed so receive traps can never take place. The RETURN from the trap routine will automatically do a SPRITE ON unless an explicit SPRITE OFF has been performed inside the trap routine. Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place this automatically disables all trapping (including ERROR, STRIG, STOP, SPRITE, INTERVAL and KEY).

Example :

```
10   ON SPRITE GOSUB 90
20   SCREEN 2, 2
30   A$ = "  "
40   FOR I = 1 TO 32
50   READ B$
60   A$ = A$ + CHR$
     (VAL ("&H" + B$))
70   NEXT
80   SPRITE $ (0) = A$
90   PUT SPRITE 0, (10, 10), 8, 0
100  PUT SPRITE 1, (110, 85), 11,
     0
110  PUT SPRITE 2, (220, 170), 6,
     0
120  GOTO 120
130  DATA 03, OF, 1F, 39, 79, FF,
     FF, FF
140  DATA 2A, 2A, 2A, 4A, 4A, 52,
     92, 92
150  DATA CO, FO, F8, 9C, 9E, FF,
     FF, FF
160  DATA 54, 54, 54, 52, 52, 4A,
     49, 49
```

4.2.1.31 ON STOP GOSUB

Purpose : To set up line numbers for BASIC to trap to when the CTRL-STOP key are pressed.

Version : Cassette, Disk

Format : ON STOP GOSUB <line number>

Remarks : When the trap occurs an automatic STOP is executed so receive traps can never take place. The RETURN from the trap routine will automatically do a STOP ON unless an explicit STOP OFF has been performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place this automatically disables all trapping (including ERROR, STRIG, STOP, SPRITE, INTERVAL and KEY).

Example :

```
10 CLS
20 ON STOP GOSUB 70
30 STOP ON
40 PRINT "x";
50 GOTO 40
60 END
70 STOP OFF
80 PRINT : PRINT "END"
90 RETURN 60
```

On pressing CTRL-STOP, line 20 renders line 70 to be executed and "END" will be printed without the "∧ C" printout.

Delete lines 20, 30 and 60 and try breaking the program by pressing CTRL-STOP.

```
10 ON STOP GOSUB 40
20 STOP ON
30 GOTO 30
40 RETURN
```

To break this program, power off the computer.

4.2.1.32 ON STRIG GOSUB

Purpose : To set up line numbers for BASIC to trap to when the trigger button is pressed.

Version : Cassette, Disk

Format : ON STRIG GOSUB list of line numbers

Remarks : When the trap occurs an automatic STRIG (n)STOP is executed so receive traps can never take place. The RETURN from the trap routine will automatically do a STRIG(n)ON unless an explicit STRIG(n) OFF has been performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place this automatically disables all trapping (including ERROR, STRIG, STOP, SPRITE, INTERVAL and KEY).

Example :

```
10 ON STRIG GOSUB 50, 60, 70
20 STRIG(0) ON :
   STRIG(1) ON :  STRIG(2) ON
30 FOR T = 1 TO 500 : NEXT
40 GOTO 30
50 PRINT "SPACEBAR" :
   STRIG(0)OFF :  RETURN
   30
60 PRINT "JOYSTICK I" :
   STRIG(1)OFF :  RETURN
   30
70 PRINT "JOYSTICK II" :
   STRIG(2)OFF :  RETURN
   30
```

Press spacebar, trigger button on joystick connected to port 1 and port 2 one at a time.

Per line 10, the program flow will be directed to line 50 as the spacebar is pressed; to line 60 as the trigger button on joystick connected to port 1; to line 70 as the trigger button on joystick connected to port 1; to line 70 as the trigger button on joystick connected to port 2. Once line 50 is executed, depressing the spacebar will not be detected. Likewise for lines 60 and 70.

4.2.1.33 OPEN

Purpose : To allocate a buffer for I/O and set the mode that will be used with the buffer.

Version : Cassette, Disk

Format : OPEN " device descriptor [filename]" [FOR mode] AS [#] filenumber

Remarks : This statement opens a device for further processing.

mode is one of the followings:

OUTPUT	: Specifies sequential output mode
INPUT	: Specifies sequential input mode
APPEND	: Specifies sequential append mode

filename : For cassette version, a string of 6 characters (maximum) is allowed. Disk file names can be a maximum of 6 characters in length with an optional character extension that is preceded by a decimal point.

filenumber is an integer whose value is between one and the maximum number of files specified in a MAXFILES statement. It is the number that is associated with the file for as long as it is OPEN and is used by other I/O statements to refer to the file.

An OPEN must be executed before any I/O may be done to the file using any of the following statements, or any statement or function requiring a filenumber:

PRINT #, PRINT # USING
INPUT #, LINE INPUT #
INPUT$, GET, PUT

Every data file is referenced by a filename and filenumber. The filename is the label you use to refer to the file. The filenumber is what the computer uses to refer to the file.

Example :

```
10  OPEN "1 : DATA" FOR OUTPUT AS
    # 1
20  A$ = "EMPLOYEE"
30  B$ = "NAME"
40  PRINT # 1, A$, B$
50  CLOSE # 1
60  C$ = "DEPARTMENT"
70  OPEN "1: DATA" FOR APPEND AS #
    1
80  PRINT # 1, C$
90  CLOSE # 1
100 OPEN "1 : DATA" FOR INPUT AS #
    1
110 LINE INPUT # 1, D1$
120 LINE INPUT # 1, C1$
130 CLOSE # 1
```

Line 10 instructs the computer to open or create a file on drive 1 called "DATA" to which we will output or write information. "#1" at the end of line 10 is the filenumber for the DATA file #1.

Line 70 - 90 reopen DATA file #1, then read in D1$ (which consists of A$ and B$) and C1$ (which consists of C$).

4.2.1.34 PAINT

Purpose : To fill in an arbitrary graphics figure of the specified fill color starting at coordinate specifier.

Version : Cassette, Disk

Format : PAINT <coordinate specifier> [, <paint color>]

Remarks : <coordinate specifier> : see the description at PUT SPRITE statement. PAINT does not allow <coordinate specifier> to be out of the screen.

<paint color> may range from 0 to 15.

PAINT can fill any figure, but painting jagged edges or very complex figures may result in an "out of memory" error. If this happens, you must use the CLEAR statement to increase the amount of stack space available. The paint color should be same as border of object.

Example :

```
10 SCREEN 1
20 COLOR 15, 4
30 LINT (50, 50) - (205, 141), 8
40 LINE (50, 141) - (205, 50), 8
50 CIRCLE (128, 96), 90, 8
60 PAINT (135, 125), 8
70 GOTO 70
```

Line 60 commands the computer to start painting at point (135, 125) using color number 8, which is magenta, till reaching a border. The final display is a circle bisected by 2 lines with its lower sector coloured.

4.2.1.35 PLAY

Purpose : To play music according to music macro language.

Version : Cassette, Disk

Format : PLAY <string expression for voice 1> [, < string expression for voice 2 > [, < string expression for voice 3>]]

Remarks : string expression for voice n is a string expression consisting of single character music commands.

PLAY implements a concept similar to DRAW by embedding a "music macro language" into a character string. When a null string is specified, the voice channel remains silent. The single character commands in PLAY are:

A, B, C, D, E, F, G [#/+][-]
Play the indicated note in the current octave. A number sign(#) or plus sign(+) afterwards indicates a sharp, a minus sign(-) indicates a flat. The #, + or - is not allowed unless it corresponds to a black key on a piano. For example, B# is an invalid note.

```
PLAY "CDEFGAB"
```

O <n>
Octave. Set the current octave for the following notes. There are 8 octaves, numbered 1 to 8. Each octave goes from C to B (CDEFGAB). Octave 4 is the default octave.

```
PLAY "O5GCAFECDGCABO5CDC"
```

Play note n. n may range from 0 to 96. n=0 means rest. This is an alternative way of selecting notes instead of specifying the octave (O n) and the note name (A-G). The C of octave 4 is 36.

```
PLAY "O4CNON36"
```

L <n>
Set the length of the following notes. The actual note length is 1/n. n may range from 1 to 64. The following table may help explain this:

LENGTH	EQUIVALENT
L1	whole note
L2	half note
L3	one of a triplet of three half notes (1/3 of a 4 beat measure)
L4	quarter note
L5	one of a quintuplet (1/5 of a measure)
L6	one of a quarter note triplet
.	
.	
.	
L64	sixty-fourth note

The length may also follow the note when you want to change the length only for the note. For example, A16 is equivalent to L16A. The default is 4.

```
PLAY "CDEFGAB L16 CDEFGAB"
```

R < n >
Pause(rest). n may range from 1 to 64, and figures of the length of the pause in the same way as L(length). The default is 4.

.
Dot or period after a note causes the note to be played as a dotted note. That is, its length is multiplied by 3/2. More than one dot may appear after the note and the length is adjusted accordingly. For example, "A..." will play 27/8 as length etc. Dots may also appear after the pause(R) to scale the pause length in the same way.

```
     PLAY "CDER2C..D..E.."
```

T < n >
Tempo. Set the number of quarter notes in a minute. n may range from 32 to 255. The default is 120.

```
     PLAY "T32 CDEFGAB T255 CDEFGAB"
```

V < n >
Volume. Set the volume of output. n may range from 0 to 15. The default is 8.

```
     PLAY "V0 CDEFGAB V15 CDEFGAB"
```

M < n >
Modulation. Set period of envelope. n may range from 1 to 65535. The default is 255.

```
     PLAY "S10 M5 CDEFGAB"
```

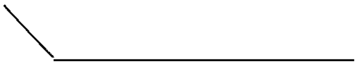
Press CTRL-STOP before typing the following command:

```
     PLAY "S10 M11115 CDEFGAB"
```

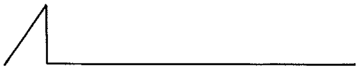
S < n >

Shape. Set shape of envelope. n may range from 1 to 15. The default is 1. The pattern set by this command are as follows:

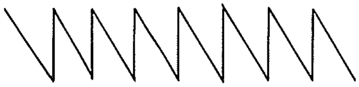
0, 1, 2, 3, 9



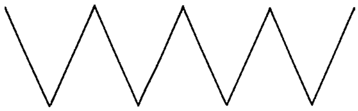
4, 5, 6, 7, 15



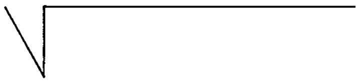
8



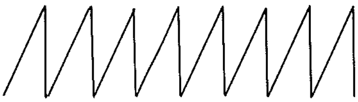
10



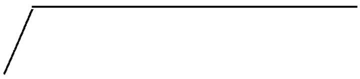
11



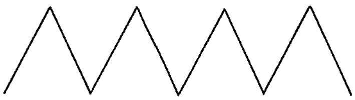
12



13



14



```
PLAY "S1 CDEFGAB"
PLAY "S15 CDEFGAB"
```

X < variable > ;
Execute a specified string.

```
10 A$ = "O4FAAAEGGGDFEDC"
20 PLAY "O4GO5CO4GECEGCGEO5CCXA$;"
```

In all of these commandss the <n > argument can be a constant like 12 or it can be "= < variable > ;" where variable is a the name of a variable. The semicolon(;) is required when you use a variable in this way, and when you use the X command. Otherwise, a semicolon is optional between commands.

Note that values specified with above commands will be reset to the system default when beep sound is generated.

Apart from the above listed functions, the computer has three seperate channels of sound that can be programmed individually to play together to create chords.
For example:

```
PLAY "O1CDE", "O3EFC", "O5GAB"
```

This command plays three notes in combination to create a chord. Also each channel can be programmed to play something entirely different from the others to create melody and harmony part of a piece of music.

Example : Enjoy the piece of music created by the below program.

```
10    ONSTOP GOSUB 410:  STOP ON
20    CLS
30    COLOR 15, 2, 2
40    SCREEN 1
50    LOCATE.5,  88:  PRINT " ******************************************"
60    LOCATE 5,  96:  PRINT " *SPECTRAVIDEO ADSR, 3 CHANNEL MUSIC DEMO*"
70    LOCATE 5, 104:  PRINT " ******************************************"
80    COLOR 15, 1, 1
90    PLAY "t60l16s0m896305", "t60l16v10m8963o3", "t60l16v1003m8963"
100   PLAY "d-4.0b-g-e-8g-B-05d-80b-g-e-8g-b-g-8g-e-", "r1r4.b", "r1r4.g-"
110   PLAY "g-8fe-m26890d-1m8963e-8g-8", "03b-2", "g-2"
120   COLOR 15, 10, 10
130   PLAY "A-4.05d-80b-8g-b-a-8o5d-8ob-4o5", "of2g-4f2:, "d-2e-8d-4.d4"
140   PLAY "e-2ob-4.b805d-4.ob-g-e-8g-b-", "b-2b-4.b8od-2e-4", "e-2"
150   PLAY "o5d-8ob-g-e-8g-b-g-8g-e-", "d-4e-4d-8o3b8", "o3b-40c4o3b-8a-8"
160   PLAY "g-8fe-d-1r8o3", "b-2.b-2", "g-4od-4d-o3bb-a-g-8fe-d-4d-o2b"
170   PLAY "b-od-e-8g-a-l8bo5d-e-g-4l16", "r8od-2.b-4", "b-a-o3b2.og-4"
180   PLAY "fe-d-4v5d-s0obb-a-", "b-4", "g-4o3"
190   PLAY "g-a-b-fe-g-d-o3bof", "e-8.fe-8.d-o3b8", "b8.od-o3b8.b-a-8"
200   PLAY "e-o3b-a-od-o3c-8.od-f-a-bo5d-f-a-", "bb-a-2.", "a-g-f-2"
210   PLAY "b8b-a-g-2b-8a-g-e-4.d-ob", "o5e-2.og-4o3b2", "oblf2"
220   PLAY "b-8a-g-e-4.e-c", "b4b-2", "g-4o2b-8r8b-4"
230   COLOR 15, 13, 13
240   PLAY "o2b-8b-o3ce-fgb-oc8e-c", "oe-2o3a-4", "o3g2f4"
250   PLAY "e-8o3b-oce-fgb-o5c8e-c", "o2b-8.o3ce-fgb-oc8e-c", "g20f4"
260   PLAY "e-8ob-o5ce-fg-b-o6e-4.", "g8o3b-oce-fg-b-b4.", "o4b-2o5b4."
270   PLAY "l24d-e-d-o5b-8a-8l16v5a-s0g--fe-l24", "o5a-8g-8b4.o", "f8e-2o"
280   PLAY "e-8d-ed-l16ob-8a-4g-a-", "b8a-8g-8a-4o3g-of", "g-8f8e-8c4b8"
290   PLAY "l8b-.g-16e-g-b-o5", "l8e-.d-16o3b-od-e-", "l8o3b-.g-16e-g-b-o"
300   PLAY "d-ob-g-e-4g-4b-.", "g-e-d-o3b-4b4oe-.", "d-o3b-g-e-4g-4b-."
310   PLAY "g-16b-o5d-e-g-b-g-", "d-16e-g-b-o5d-e-d-", "g-16b-od-e-g-b-g-"
320   PLAY "ob-4d-4r2o6d-4v4d-4.s016", "o3c-4a-4og-1g-", "o3g-4f4oe-1e-"
330   PLAY "o5b-g-e-8g-b-o6d-8o5b-g-e-8g-b-g-8g-e-", "g-g-1", "e-e-1"
340   PLAY "lg-fe-d-.ol8bb-a-l16", "b-2.d-2.l16", "g-1.l16o3"
```

```
350   PLAY "g-a-b-fe-a-g-d-o3bof", "e-8.fe-8.d-o3b8", "b8.od-o3b8.b-a-8"
360   PLAY "e-o3b-a-od-e-o3b-bo", "bb-a-8.b-b", "a-g-f8.g-a-"
370   PLAY "d-e-o3b-a-o", "b8b-a-", "a-8g-f-"
380   PLAY "d-e-o3b-bog-a-bo5d-e-g-a-", "a-8oe-8d-e-g-a-bo5d-e-", "f8g-a-2"
390   PLAY "112bo6D-E-O", "L12G-A-BB", "A-4A-24"
400   PLAY "s0m53780o9-1.", "s0o6r16d-1.", "s0o5r32b-1."
410   COLOR 15, 4, 5
```

4.2.1.36 PRINT
PRINT # USING

Purpose : To write data to the specified channel.

Version : Cassette, Disk

Format : PRINT # <filenumber> , <expression>
PRINT # <filenumber> , USING <string expression> ; <list of expression>

Remarks : See PRINT/PRINT USING statements for details.

Example :

```
10 OPEN "1 : RECORD" FOR OUTPUT AS
   # 1
20 A$ = "AMT1" : B$ = "AMT 2" :
   C$ = "AMT3"
30 A = 12.235 : B = 64.2 :
   C = 129.653
40 PRINT # 1, A$, B$, C$
50 PRINT # 1, USING "$$###.##";
   A, B, C
60 CLOSE # 1
70 OPEN "1 : RECORD" FOR INPUT AS #
   1
80 LINE INPUT # 1, D$
90 CLOSE # 1
```

Line 20 and 30 define the variables. Lines 40 and 50 instruct the computer to write them on the disk. The command "PRINT # 1, USING" writes numeric data to disk without explicit delimiters. The comma at the end of the format string serves to seperate the items in the disk file.

4.2.1.37 PSET PRESET

Purpose : To set/reset the specified coordinate. For the detail of the coordinate specifier , see the description at PUT SPRITE statement.

Version : Cassette, Disk

Format : PSET <coordinate specifier> [, <color>]
PRESET <coordinate specifier [, <color>]

Remarks : <coordinate specifier> : coordinates for drawing or setting a dot; may be either absolute or relative.

<color> : integer from 0 to 15 which assigns dot color.

PSET/PRESET draws a dot at the assigned position on the screen.

PRESET allows the attribute argument to be left off and it is defaulted to foreground color.

The only difference between PSET and PRESET is that if no <color> is given in PRESET statement, the background color is selected. When a <color> argument is given, PRESET is identical to PSET.

If the <coordinate specifier> is out of range, no action is taken and an error is given. If <color> is larger than 15 then this will result in an illegal function call.

Example

```
10 SCREEN 2
20 FOR Y = 60 TO 120
30 PSET (100, Y), 6
40 FOR I = 1 TO 20 : NEXT
50 PRESET (100, Y)
60 NEXT
```

The appearance and disappearance of a point creates the impression of motion from top to bottom of the screen. Line 40 is merely a time delay.

4.2.1.38 PUT

Purpose : To write a record from a random buffer to a random disk file.

Version : Disk

Format : PUT[#] <filenumber> [, <record number>]

Remarks : <filenumber> is the number under which the file was OPENed. If <record number> is omitted, the record will have the next available record number after the last PUT. The largest possible record number is 32767. The smallest record number is 1.

PRINT# and PRINT# USING may be used to put characters in the random file buffer before a PUT statement.

Any attempt to read or write past the end of the buffer causes "Field overflow" error.

Example :

```
10 INPUT "DATE:"; D$
20 INPUT "DEMO:"; M$
30 OPEN "1: MEMO" AS #1
40 FIELD #1, 10 AS D$, 20 AS M$
50 LSET A$ = D$
60 LSET B$ = M$
70 PUT #1, 20
80 CLOSE #1
```

A random file named "MEMO" is opened to have data written into it. Line 70 writes the data from the buffer to the diskette.

4.2.1.39 PUT (graphics)

Purpose : To output graphic patterns in the assigned position on the screen.

Version : Cassette, Disk

Format : PUT (x, y), < array name > , [, < operation >]

Remarks : (x, y): coordinates of the upper left-hand corner of the rectangular region on the screen.

<array name> : name of numerical array containing graphic pattern being output on the screen.

<operation> : assignment of operation to be performed with data already displayed on the screen when a graphic pattern is output on the screen. Operations include PSET, PRESET, XOR, OR and AND. If omitted, it is interpreted as XOR.

In contrast to GET, PUT causes the array data to be output on the screen. For<operation>, the following may be selected.

PSET: Output the graphic pattern contained in the array on the screen as is (opposite operation from GET).

PRESET: Reverse the graphic pattern contained in the array and output it on the screen (similar to a photographic negative).

AND: The result of combining the graphic pattern contained in the array and the data already displayed on the screen on a one to one basis using AND is output on the screen.

OR: The graphic pattern in the array is output on the screen overlapping the data already displayed there.

Example :

```
10  SCREEN 1
20  DEFINT C
30  CIRCLE (128, 96), 8
40  LINE (128, 104) - (112, 118)
50  LINE - (144, 118)
60  LINE - (128, 104)
70  DIM C (30, 35)
80  GET (110, 80) - (160, 120), C
90  PUT (20, 16), C, PSET
100 PUT (40, 16), C, PSET
110 PUT (60, 16), C, PSET
120 PUT (80, 16), C, PSET
130 GOTO 130
```

Lines 30 through 60 represent the figure drawn. Line 70 creates a rectangular array large enough to hold it. After getting the image on line 80, it is put to different locations as specified on lines 90 through 120.

4.2.1.40 PUT SPRITE

Purpose : To set up sprite attributes.

Version : Cassette, Disk

Format : PUT SPRITE <sprite plane number> [, <coordinate specifier>] [, <color>] [, <pattern number>]

Remarks : <sprite plane number> ranges from 0 to 31.

<coordinate specifier> always can come in one of two forms:

```
STEP ( x offset, y offset) or
( absolute x, absolute y)
```

The first form is a point relative to the most recent point referenced. The second form is more common and directly refers to a point without regard to the last point referenced. Examples are:

(10, 10)	Absolute form
STEP (10,0)	Offset 10 in x and 0 in y
(0, 0)	Origin

Note that when BASIC scans coordinate values it will allow them to be beyond the edge of the screen, however values outside the integer range (-32768 to 32767) will cause an overflow error. And the values outside the screen will be substituted with the nearest possible value. For example, 0 for any negative coordinate specification.

Note that (0,0) is always the upper left hand corner. It may seem strange to start numbering y at the top so the bottom left corner is (0,191) in high-resolution, but this is the standard.

Above description can be applied wherever graphic coordinate is used.

x coordinate: x may range from -32 to 255. y coordinate: y may range from -32 to 191. If 208 (&HD0) is given to y, all sprite planes behind disappear until a value other than 208 is given to that plane. If 209 (&HD1) is specified to y, then that sprite disappears from the screen. Refer to VDP (Video Display Processor) manual for further details. Thus to erase a sprite, set y to 209. To erase all the sprites following a specific < sprite-plane > , set the y value to 208.

When a field is omitted, the current value is used. At start up, color defaults to the current foreground color.

< pattern number > specifies the pattern of the sprite, and must be less than 256 when size of sprites is 0 or 1, and must be less than 6 when size of sprites is 2 or 3. < pattern number> defaults to the < sprite plane number > . See also SCREEN statement and SPRITE$ variable.

Example

```
10   FOR I = 1 TO 8
20   READ A$
30   B$ =B$ + CHR$ (VAL
     ("&B" + A$))
40   NEXT
50   SCREEN 1, 1
60   SPRITE$ (0) = B$
70   PUT SPRITE 0, (128, 96),
     15, 0
80   GOTO 80
90   DATA 0
100  DATA 00111100
110  DATA 00100000
120  DATA 00111100
130  DATA 00000100
140  DATA 00000100
150  DATA 00111100
160  DATA 0
```

What you see on the screen is the character "S" in white.

Lines 90 to 160 specify the sprite's shape. There are 8 characters on each data statement.
The zeros make the display transparent at that point of the shape while the ones are the points lit up.

Lines 10 to 40 set up a loop to read data.
Line 30 converts data into binary strings, appending each one to the previous string and store this shape unit in B$.

Line 60 picks sprite numbered 0. It carries the shape contained in B$.

Line 70 puts the sprite#0 that is specified in line 60 on plane 0 at position (128, 96) using color #15.

Sprites are not limited to 8 by 8 pixels. They can be placed within a 16 by 16 box.

When SCREEN size 0 or 1 is selected, the sprite size is limited to 8 by 8. If size 2 is selected, the use of 16 by 16 box is allowed. The following example illustrates how the 16 by 16 box is filled:

```
10  SCREEN 1, 3
20  FOR X = 1 TO 32
30  READ A$
40  RESTORE
50  S$ = S$ + CHR$ (VAL ("&B" +
    A$))
60  SPRITE $(0) = S$
70  PUT SPRITE 0, (128, 96),
    15, 0
80  NEXT
90  GOTO 90
100 DATA 11110001
```

Notice that the computer first fills an 8 x 16 box and then fills the adjacent 8 x 16 box to make a 16 x 16 box.

4.2.1.41 SAVE

Purpose	:	To save a BASIC program file to the specified device.
Version	:	Cassetee, Disk
Format	:	SAVE " [<device descriptor>] <filename> " [,A,]

Remarks : <device descriptor> : For cassette, this can either be "CAS:" or simply omit this part. For disk drive, it should be "1:" or "2:" depending on which disk drive is in use.
<filename> : For cȁssette version, a string of 6 characters (maximum) is allowed. Disk filenames can be a maximum of 6 characters in length with an optional character extension that is preceded by a decimal point. The maximum number of characters in an extension is 3. If the file name is more than 6 characters, BASIC inserts a decimal point after the sixth character and uses the next three characters as an extension. Any additional points are ignored.

When saving to cassette, its motor is turned on and the file is written to the tape.

If a file with the same filename alread exists on the diskette, it will be written over.

The "A" option saves the program in ASCII format. Otherwise, BASIC saves the file in a compressed binary (tokenized) format. ASCII files take more space. Some types of access require that files be in ASCII format. For example, a file intended to be merged must be saved in ASCII format.

Example

```
  SAVE "CARACE"
```

Save the program in memory on tape under the filename "CARACE".

```
  SAVE "1 : SYSGEN.BAS"
```

The program SYSGEN.BAS is saved on the diskette in drive 1.

4.2.1.42 SCREEN

Purpose : To assign the screen mode and sprite size.

Version : Cassette, Disk

Format : SCREEN [<mode>] [,<sprite size>]

Remarks : <mode>

0:	40 x 24	text mode
1:	256 x 192	high resolution mode
2:	64 x 48	low resolution mode

<sprite size> determines the size of sprite.

0:	8 x 8	unmagnified
1:	8 x 8	magnified
2:	16 x 16	unmagnified
3:	16 x 16	magnified

Example :

```
10  SCREEN 0, 1
20  LOCATE 10, 10 : PRINT
    "BEETHOVEN"
30  FOR I = 1 TO 500
40  NEXT
50  SCREEN 1.1
60  LOCATE 80, 80 : PRINT
    "BEETHOVEN"
70  FOR I = 1 TO 500
80  NEXT
90  SCREEN 2, 1
100 LOCATE 20, 60 : PRINT
    "BEETHOVEN"
110 GOTO 110
```

This example demonstrates the different printout effect of three modes. Although the characters printed on screen by lines 20 and 60 are very similar, notice the different locations of cursor position.

As the screen mode is changed to 2
in line 90. The printout characters
are much larger.

4.2.1.43 SOUND

Purpose : To write value directly to the register of PSG

Version : Cassette, Disk

Format : SOUND <register of PSG> , <value to be written>

Remarks : <register of PSG>: PSG (Programmable Sound Generator) has 13 available registers (1 to 13). <value to be written> ranges from 1 to 255.

Basically, the following blocks in the PSG produce some programmed sounds:

Tone Generators
Produce the basic square wave tone frequencies for channel A, B, C.

Noise Generator
Produce a frequency modulated pseudo random pulse width square wave output.

Mixers
Combine the outputs of the Tone Generators and the Noise Generator. One for each channel (A, B, C).

Amplitude control
Provide the D/A Converters with a fixed or variable amplitude pattern. The fixed amplitude is under direct CPU control; the variable amplitude is accomplished by using the output of the Envelope Generator.

Envelope Generator
Produce an envelope pattern which can be used to amplitude modulate the output of each Mixer.

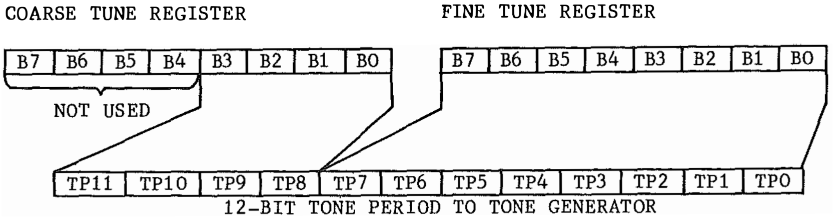
D/A Converters
Each of the three D/A Converters produces up to a 16 level output signal as determined by the Amplitude Control.

PROGRAMMABLE SOUND GENERATOR REGISTERED ARRAY
(14 READ/WRITE CONTROL REGISTERS)

REGISTER	BIT	B7	B6	B5	B4	B3	B2	B1	B0
R0	Channel A Tone Period	8-Bit Fine Tune A							
R1						4 Bit Coarse Tune A			
R2	Channel B Tone Period	8-Bit Fine Tune B							
R3						4 Bit Coarse Tune B			
R4	Channel C Tone Period	8-Bit Fine Tune C							
R5						4 Bit Coarse Tune			
R6	Noise Period				5-Bit Period Control				
R7	Mixer Control - I/O Enable	Input Enable		Noise Enable			Tone Enable		
		I/OB	I/OA	C	B	A	C	B	A
R8	Channel A Amplitude				M	L3	L2	L1	L0
R9	Channel B Amplitude				M	L3	L2	L1	L0
R10	Channel C Amplitude				M	L3	L2	L1	L0
R11	Envelope Generator Control	8-Bit Fine Tune E							
R12		8-Bit Coarse Tune E							
R13	Envelope Shape/Cycle					E3	E2	E1	E0

Tone Generator Control

The PSG has 3 tone channels A, B and C. The frequency for each channel is obtained by counting down the input clock by 16 times the programmed 12-bit Tone Period value. Each 12-bit value is obtained by combining the contents of the relative Coarse and Fine Tune registers, as illustrated in the following:



The following equations describe the relationship between the desired output tone frequency and the input clock frequency and Tone Period:

$$\langle \text{desired tone frequency} \rangle = \left\langle \frac{\text{input clock frequency}}{16 * \langle \text{tone period} \rangle} \right\rangle$$

$$\langle \text{tone period} \rangle = 256 * \langle \text{coarse tune value} \rangle + \langle \text{fine tune value} \rangle$$

The above values should be calculated on decimal basis.

CHANNEL	COARSE TUNE REGISTER	FINE TUNE REGISTER
A	R1	RO
B	R3	R2
C	R5	R4

For example :

```
10 INPUT "ENTER FREQUENCY"; A
20 F = 3579545 / (16* A)
30 H = F / 256
40 L = F AND 255
50 SOUND 0, L
60 SOUND 1, H
70 SOUND 8, 15 : PRINT "VOLUME
   CONTROL OF CHANNEL A"
80 SOUND 7 , 254 : PRINT "&B 11111110
   TO ENABLE CHANNEL A"
```

Noise Generator Control

The frequency of the noise source is obtained by counting down the input clock by 16 times the result by the programmed 5-bit Noise Period value.

```
R6   | B7  B6  B5 | B4  B3  B2  B1  B0 |
      \____ ____/
           V
      NOT USED     5-BIT NOISE PERIOD
                   TO NOISE GENERATOR
```

The noise frequency equation is

$$\langle \text{desired noise frequency} \rangle = \frac{\langle \text{input clock frequency} \rangle}{16 * \langle \text{noise period} \rangle}$$

Amplitude Control

The amplitudes of the signals generated by each of the three D/A Coverters (one each of channels A, B and C) is determined by the contents of the lower 5 bits (B4 to BO) of registers R8, R9 and R1O.

The amplitude "mode" (bit M) selects either fixed level amplitude (M=O) or variable level ampltiude (M=1). Bits L3 to LO, defining the value of a fixed level ampltiude are only active when M=O. Value for volume ranges from O to 15 with 15 being the loudest. When M=1, the amplitude of each channel is determined by the envelope pattern as defined by the Envelope Generator's 4 bit output E3 to E0.

The amplitude mode (bit M) can be regarded as an "envelope enable" bit. When M=0 the envelope is not used and when M=1 the envelope is enabled.

Value for volume ranges from 0 to 15 with 15 being the loudest.

For example :

```
10 SOUND 0, 100
20 SOUND 1, 0
30 SOUND 7, 254 :  REM TURN ON
   CHANNEL A (MIXER)
40 FOR I = 15 TO 0 STEP -1
50 SOUND 8, I
60 FOR J = 1 TO 200 : NEXT J : REM
   DELAY
70 NEXT I
```

This program renders a high pitched sound fading away because of the decrement of volume from 15 to 0.

The amplitude control register can also be used to direct the envelope period of each channel, by setting the amplitude channel to a value of 16 (&B 10000), the amplitude of the corresponding channel will be controlled by register 11, 12 and 13. See the Envelope Period Control for details.

Mixer Control

Register 7 is a multi-function Enable register which controls the three Noise/Tone Mixers and the two general purpose I/O Ports.

The Mixer combines the noise and tone frequencies for each of the three channels. The determination of combining neither, either or both noise and tone frequencies on each channel is made by the state of bit 0 to bit 5 of register 7. The direction (input or output) of the two general purpose I/O Ports (I/OA and I/OB) is determined by the state of bits B7 and B6 of R7, which are ignored by BASIC. For the bit logical value : 1 disables the channel while 0 enables it.

Disabling noise and tone does not turn off a channel. Turning a channel off can only be accomplished by writing all zeros into the corresponding Amplitude Control register R8, R9 or R10.

For example :

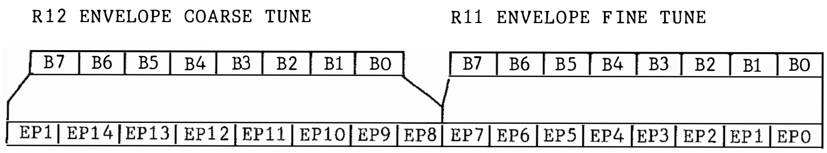
```
   SOUND 7, &B 11111110
 Turn on tone channel A.
   SOUND 7, &B 11110110
 Enable both noise and tone channel A.
```

Envelope Generator Control

To accomplish the generation of fairly complex envelope patterns, two independent methods of control are provided in the PSG: first, it is possible to vary the frequency of the envelope using registers R11 and R12; and second, the relative shape and cycle pattern of the envelope can be varied using register R13.

Envelope Generator Control

The frequency of the envelope is obtained by first counting down the input clock by 256, then by further counting down the result by the programmed 16-bit Envelope Period value, which is obtained by combining the contents of the Envelope Coarse and Fine Tune registers.



16-BIT ENVELOPE PERIOD TO ENVELOPE GENERATOR

The 16-bit value programmed in the combined Coarse and Fine Tune registers is a period value - the higher the value in the registers, the lower the resultant envelope frequency.

The envelope frequency equations are:

$$\langle \text{desired envelope frequency} \rangle = \frac{\langle \text{input clock frequency} \rangle}{256 * \langle \text{envelope period} \rangle}$$

$$\text{envelope period} = 256 * \langle \text{coarse tune register value} \rangle + \langle \text{fine tune register value} \rangle$$

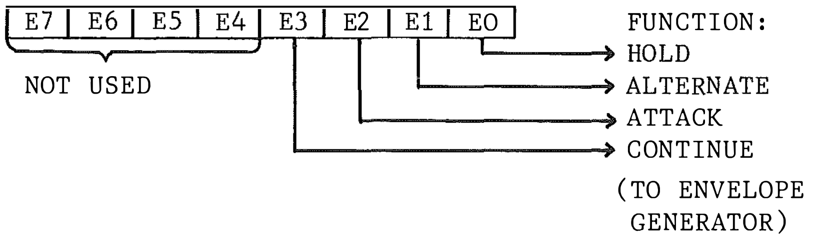
Envelope Shape/Cycle Control (R13)

The Envelope Generator further counts down the envelope frequency by 16, producing a 16-state per cycle envelope pattern as defined by its 4-bit counter output E3 E2 E1 EO.

The shape and cycle pattern of any desired envelope is accomplished by controlling the count pattern (count up/count down) of the 4-bit counter and by defining a single-cycle or repeat-cycle pattern.

This envelope shape/cycle control is contained in the lower 4-bits (B3 - BO) of register R13.

R13 ENVELOPE SHAPE/CYCLE CONTROL REGISTER



The definition of each function is as follows:

Hold

When set to logic "1", limits the envelope to one cycle, holding the last count of the envelope counter (E3 - EO).

Alternate

When set to logic "1", the envelope counter reverses count direction (up-down) after each cycle. When both the Hold bit and the Alternate bit are ones, the envelope counter is reset to its initial count before holding.

Attack

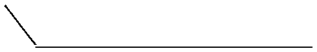
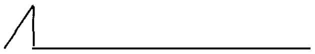
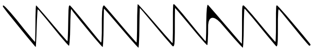
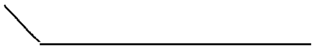
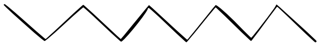
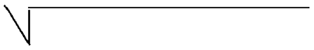
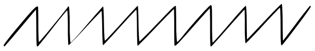
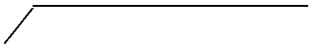
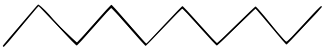
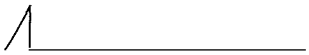
When set to logic "1", the envelope counter will count up (attack) from E3 E2 E1 EO = 0000 to E3 E2 E1 EO = 1111; when set to logic "0", the envelope counter will count down (decay) from 1111 to 0000.

Continue

When set to logic "1", the cycle pattern will be as defined by the Hold bit; when set to logic "0", the envelope generator will reset to 0000 after one cycle and hold at that count.

Given below is a chart of the binary count sequence of E3 E2 E1 EO for each combination of Hold, Alternate, Attack and Continue. When selected by the Amplitude Control registers, these outputs are used to amplitude modulate the output of the Mixers.

ENVELOPE SHAPE/CYCLE CONTROL

R13 BITS				GRAPHIC REPRESENTATION OF ENVELOPE GENERATOR OUTPUT E3 E2 E1 EO	SELECTED VALUE
B3	B2	B1	BO		
CONTINUE	ATTACK	ALTERNATE	HOLD		
0	0	X	X		0, 1, 2, 3
0	1	X	X		4, 5, 6, 7
1	0	0	0		8
1	0	0	1		9
1	0	1	0		10
1	0	1	1		11
1	1	0	0		12
1	1	0	1		13
1	1	1	0		14
1	1	1	1		15

ENVELOPE PERIOD
(DURATION OF ONE CYCLE)

Now try the following example:

```
10   SOUND 0, 100
20   SOUND 1, 0  :  REM TONE CHANNEL A
30   SOUND 7, &B 11111110:  REM ENABLE A
40   SOUND 8, 16  :  REM ENABLE REG 11, 12
50   SOUND 13, 14  :  REM SHAPE SELECT
60   S = .5  :  REM FREQ =  .5HZ
70   CLOK = 3579545
80   L = CLOK / (256* S) AND 255
90   H = CLOK / (256* S) /256
100  SOUND 11, L
110  SOUND 12, H
120  END
```

4.2.1.44 SPRITE ON/OFF/STOP

Purpose	: To activate/deactivate trapping of sprite in a BASIC program.
Version	: Cassette, Disk
Format	: SPRITE ON/OFF/STOP
Remarks	: A SPRITE ON statement must be executed to activate trapping of sprite. After SPRITE ON statement, if a line number is specified in the ON SPRITE GOSUB statement then every time BASIC starts a new statement it will check to see if the sprites coincide. If so it will perform a GOSUB to the line number specified in the ON SPRITE GOSUB statement. If a SPRITE OFF statement has been executed, no trapping take place and the event is not remembered even if it does take place. If a SPRITE STOP statement has been executed, no trapping will take place, but if the sprites coincide this is remembered so an immediate trap will take place when SPRITE ON is executed.
Example	: Refer to ON SPRITE GOSUB.

4.2.1.45 STOP ON/OFF/STOP

Purpose	:	To activate/deactivate trapping of a CTRL-STOP.
Version	:	Cassette, Disk
Format	:	STOP ON/OFF/STOP
Remarks	:	A STOP ON statement must be executed to activate trapping of a CTRL-STOP After STOP ON statement, if a line number is specified in the ON STOP GOSUB statement then every time BASIC starts a new statement it will check to see if a CTRL-STOP was pressed. If so, it will perform a GOSUB to the line number specified in the ON STOP GOSUB statement. If a STOP OFF statement has been executed, no trapping takes place and the event is not remembered even if it does take place. If a STOP STOP statement has been executed, no trapping will take place, but if a CTRL-STOP is presed this is remembered so an immediate trap will take place when STOP ON is executed.
Example	:	Refer to ON STOP GOSUB.

4.2.1.46 STRIG ON/OFF/STOP

Purpose	:	To activate/deactivate trapping of trigger buttons of joysticks in a BASIC program.
Version	:	Cassette, Disk
Format	:	STRIG (< n >) ON/OFF/STOP
Remarks	:	n can be in the range of 0 to 2. If n =0, the space bar is used for a trigger button. If n is 1, the trigger of joystick 1 is used. When n is 2, joystick 2 is referenced. A STRIG(n)ON statement must be executed to activate trapping takes place and the event is not remembered even if it does take place. If a STRIG(n)OFF statement is executed, trapping will be disabled and the check on the trigger status will be suspended. Also it will not be maintained even if it is depressed. If a STRIG(n)STOP statement is executed, no trapping will take place, but if the trigger button is pressed this is remembered so an immediate trap will take place when STRIG(n)ON is executed.
Example	:	Refer to ON STRIG GOSUB.

4.2.1.47 VPOKE

Purpose : To poke a value to a specified location of VRAM.

Version : Cassette, Disk

Format : VPOKE < address of VRAM > , < value to be written >

Remarks : < address of VRAM > can be in the range of 0 to 16383.

Example :

```
10 VPOKE &H3000, &H10
20 B = VPEEK (&H3000)
30 PRINT B
RUN
16
Ok
```

Line 10 writes the value &H10 into VRAM at the location &H3000. Line 20 reads this value back. It is printed in decimal figure by line 30.

4.2.1.48 WIDTH

Purpose : Set the width of display during text mode.

Version : Cassette, Disk

Format : WIDTH < width of screen in text mode >

Remarks : < width of screen in text mode>is a valid numeric expression returning an integer either 39 or 40. The default is 39. If the 80 - column interface cartridge and monitor are installed, then width 80 is also valid as disk BASIC is run.

To set the printed width in the number of characters for the screen. Changing the screen width causes the screen to be cleared.

Example :

```
10 FOR I = 1 TO 50
20 PRINT I;
30 NEXT
```

Try the above example by typing:

```
RUN
```

Then enter

```
WIDTH 40
```

and notice the screen is cleared and Ok prompt appears on the top left hand screen. Type RUN to execute the program. Notice one more character is printed per row for the second program.

Try the following if 80 - column interface card is installed and the monitor is turned on:

```
WIDTH 80
```

Enter and run the above program again.

4.2.2 Functions

4.2.2.1 CVI, CVS, CVD

Purpose : Convert string values to numeric values.

Version : Disk

Format :

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

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

Example :

```
  .
  .
  .
 70 FIELD #1,4 AS N$, 12 AS B$,...
 80 GET #1, 20
 90 Y = CVS (N$)
  .
  .
  .
```

The CVS command converts the string variable N$ into a single precision value which is stored in container "Y".

4.2.2.2 EOF

Purpose : Indicate an end of file condition.

Version : Cassette, Disk

Format : EOF (< filenumber >)

Remarks : < filenumber > is the number specified on the OPEN statement.

Return -1 (true) if the end of a sequential file has been reached. Otherwise, returns 0. Use EOF to test for end-of-file while inputing to avoid "Input past end" error.

Example :

```
10  OPEN "1 : DEMO" FOR OUTPUT AS
    # 1
20  FOR A = 0 TO 50
30  PRINT # 1, A
40  NEXT A
50  CLOSE # 1
60  OPEN "1 : DEMO" FOR INPUT AS #
    2
70  IF EOF(1) THEN GOTO 110
80  INPUT # 1, A
90  GOTO 70
100 CLOSE # 1
110 END
```

This program writes the number 0 to 50 into a file and then reads them back.

On line 70, the EOF function tests to see whether or not the end of a file is reached. If the end of a file is reached, then EOF returns to the program is 1. A 0 will be returned if the end of the file has not been reached.

4.2.2.3 LOC

Purpose	:	Return the record number under which information is stored.
Version	:	Disk
Format	:	LOC (< filenumber >)
Remarks	:	With random disk files, LOC returns the record number just read or written from a GET or PUT. If the file was opened but no disk I/O has been performed yet, LOC returns a 0. With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was OPENed.
Example	:	`200 IF LOC (1) > 50 THEN STOP`

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

This checks whether the last record number just read from a GET command or written to by a PUT command exceeds 50. If this is true then the program execution halts.

4.2.2.4 MKI$, MKS$, MKD$

Purpose : Convert numeric values to string values.

Version : Cassette, Disk

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

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

Example :

```
90  AMT = K + T
100 FIELD #1,8 AS D$,20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1, 10
 .
 .
```

Line 110 uses the MKS$ command to convert the numeric data stored in "AMT" into a string variable called D$.

4.2.2.5 PAD

Purpose : Return various status of touch pad.

Version : Cassette, Disk

Format : PAD (< n >)

Remarks : < n > ranges from 0 to 3.

For < n > = 0, the status of touch pad is returned; -1 when touched, 0 when released.

For < n > = 1, the x-coordinate is returned, ranging from 0 to 255.

For < n > = 2, the y-coordinate is returned, ranging from 0 to 255.

For < n > = 3, the status of switch on the pad is returned; -1 when pressed, 0 otherwise.

Example :

```
10   SCREEN 2
20   COLOR 15, 5, 5
30   CLS
40   IF PAD (0) THEN 60
50   GOTO 40
60   X = PAD(1) : Y = PAD(2)
70   PSET (X, Y)
80   IF PAD(0) THEN 100
90   GOTO 80
100  X = PAD(1) : Y = PAD(2)
110  LINE - (X, Y)
120  GOTO 80
```

Geometrical diagram drawn on the tablet will be displayed on the screen.

Lines 40 and 80 check the status of tablet.

Line 60 and 100 get the points specified by the user.

4.2.2.6 POINT

Purpose : Return the color of the specified point on the screen.

Version : Cassette, Disk

Format : POINT (X, Y)

Remarks : x, y are the coordinates of the point to be used. The coordinates must be in absolute form.

Return the color of a specified pixel. If the point is out of range the value -1 is returned.

Example :

```
10 SCREEN 1
20 COLOR 15, 4, 5
30 PSET (200, 100), 8
40 A = POINT (200, 100)
50 B = POINT (100, 100)
60 C = POINT (200, 200)
70 SCREEN 0
80 PRINT A; B; C;
```

Line 40 sets A to be the color of point (200, 100). Likewise for lines 50 and 60. As the program is executed, the values for A, B, C are found to be 8, 5 and -1 respectively.

The color number (8) for point (200, 100) is specified on line 30.

The color number (5) for point (100, 100) is determined on line 20.

If the point given is out of range, eg. (200, 200), the value -1 is returned.

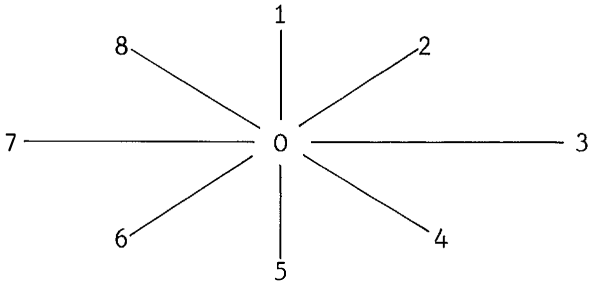
4.2.2.7 STICK

Purpose : Return the directions of a joystick.

Version : Cassette, Disk

Format : STICK (< n >)

Remarks : < n > can be in the range of 0 to 2. If < n > = 0, the cursor key is used as a joystick. If < n > is either 1 or 2, the joystick connected to proper port is used. When neutral, 0 is returned. Otherwise, value corresponding to direction is returned.



Example :

```
10   SCREEN 0:
20   X% = 20 : Y% = 12
30   LOCATE X%, Y% : PRINT "*"
40   S = STICK (0)
50   IF S = 0 OR S = 1 OR S = 5
     THEN 40
60   LOCATE X%, Y% : PRINT "  "
70   ON (S + 1)/4  GOTO 80, 100
80   X% = X% + 1 : IF X% = 39 THEN
     X% = 0
90   GOTO 30
100  X% = X% - 1 : IF X% = -1 THEN
     X% = 38
110  GOTO 30
```

This program demonstrates the movement of a character across the screen.

On line 40, movement of cursor key is recorded per container S.

Line 50 restricts its movement to be left/right only.

4.2.2.8 STRIG

Purpose	:	Return the status of a trigger button of a joystick.
Version	:	Cassette, Disk
Format	:	STRIG (< n >)
Remarks	:	< n > can be in the range of 0 to 2. If<n> = 0, the space bar is used as a trigger button. If < n > is 1, the trigger of a joystick 1 is used; when < n > is 2 joystick 2. 0 is returned if the trigger is not being pressed. Otherwise, -1 is returned.

Example :

```
10 CLS
20 COLOR 15
30 IF STRIG(0) THEN GOSUB 60
40 PRINT "*"
50 GOTO 30
60 PRINT "#"
70 RETURN 30
```

A pattern of "*" and "#" is printed on the screen as this program is executed.

Per line 30, on pressing space bar "#" will be printed instead of "*".

4.2.2.9 VPEEK

Purpose : Return a value of VRAM.

Version : Cassette, Disk

Format : VPEEK (< address of VRAM >)

Remarks : < address of VRAM > can be in the range of 0 to 16383.

The returned value lies within the range 0 to 255.

Refer to VPOKE which is the complementary function.

Example :

```
10 VPOKE &H3000, &H22
20 B = VPEEK (&H3000)
30 PRINT B
RUN
34
Ok
```

Line 20 reads the value of the byte stored in user-assigned hex offset memory location 3000 (12288 bytes), which is 22 in hex number (or 34 in decimal integer).

4.2.3 Special variables

The followings are special variables. When assigned, the content is changed, when evaluated, the current value is returned.

4.2.3.1 SPRITE$

Purpose : The pattern of sprite.

Version : Cassette, Disk

Format : SPRITE$ (< pattern number >)

Remarks : < pattern number > must be less than 256 when size of sprites is 0 or 1, less than 64 when size of sprites is 2 or 3.

The length of this string variable is fixed to 32 bytes. So, if assign the string that is shorter than 32 characters, the chr$(0)s are added.

Example :

```
10   SCREEN 1
20   FOR I = 1 TO 8
30   READ B $
40   A$ = A$ + CHR$
     (VAL ("&B" + B$))
50   NEXT I
60   SPRITE $ (1) = A$
70   SPRITE $ (2) = A$ + A$
80   SPRITE $ (3) = A$ + A$ + A$
90   SPRITE $ (4) = A$ + A$ + A$
     +A$
100  PUT SPRITE 1, (40, 60), 15, 1
110  PUT SPRITE 2, (80, 70), 8, 2
120  PUT SPRITE 3, (120, 80), 10, 3
130  PUT SPRITE 4, (160, 90), 1, 4
140  GOTO 140
150  DATA 00111100
160  DATA 01000010
170  DATA 01000010
180  DATA 00111100
190  DATA 01000010
200  DATA 10000001
210  DATA 10000001
220  DATA 01111110
```

Four geometrical figures, each built up by the numeral "8", appear on the screen.

Line 30 reads information from the data lines.

Line 40 assigns it to the container A$.

The eight lines of data (150 to 220) provide the shape you wish to put on the screen. Line 40 converts the data code into binary strings which consist of ones and zeros. Then each piece of the shape are fitted together. The whole outfit is stored in container A$.

Line 60 creates a sprite which is numbered as 1. Three other sprites are generated on lines 70, 80 and 90.

Let's have a closer look at line 100. It reads as: put the sprite which is specified at the end of the line (i.e. 1) and place it on surface numbered 1 at position (40, 60) in color number 15.

There is a more elegant way of creating the same effect. Instead of reading and writing data in a FOR NEXT loop and data lines, all data are written in on a line. Also hex figures can be used instead of binary numbers. Try the following program:

```
10   SCREEN 1
20   A$ = CHR$(&H3C) + CHR$(&H42)
     + CHR$(&H42) + CHR$(&H3C)
     + CHR$(&H42) + CHR$(&H81)
     + CHR$(&H81) + CHR$(&H7E)
30   SPRITE $ (1) = A$
40   SPRITE $ (2) = A$ + A$
50   SPRITE $ (3) = A$ + A$ + A$
60   SPRITE $ (4) = A$ + A$ + A$
     + A$
70   PUT SPRITE 1, (40, 60), 15, 1
80   PUT SPRITE 2, (80, 70), 8, 2
90   PUT SPRITE 3, (120, 80), 10, 3
100  PUT SPRITE 4 , (160, 90), 1, 4
110  GOTO 110
```

4.2.3.2 TIME

Purpose : System intend timer.

Version : Cassette, Disk

Format : TIME

Remarks : An unsigned integer. TIME is automatically incremented by 1 every time VDP generates interrupt (60 times per second), thus when an interrupt is disabled (for example, when manipulating cassette), it retains the old value.

Example :

```
10   DEFINT H-S
20   TIME = 0
30   T = TIME/60
40   H = T/3600
50   M = (T - 3600*M)/60
60   S = T - 60*M - 3600*H
70   SCREEN 2
80   PRINT H":" M":" S
90   PRINT TIME
100  GOTO 30
```

This program serves as a clock.

Notice that the value of TIME increments by 60 then S will increment by 1.

4.2.4 Machine dependent statements and functions

4.2.4.1 INP

Purpose	:	Return the byte read from the port.
Version	:	Cassette, Disk
Format	:	INP (< port number >)
Remarks	:	< port number > lies in the range 0 to 255. INP is the complementary function to the OUT statement.
Example	:	X = INP (250) This instruction reads a byte from port 250 and assigns it to the variable X.

4.2.4.2 OUT

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

Version : Cassette, Disk

Format : OUT < port number > , < integer expression >

Remarks : < port number > lies in the range 0 to 255.
< integer expression > is the data to be transmitted within the range 0 to 255.

OUT is the complementary statement to the INP function.

Example :

```
  OUT 32, 100
```

This sends the value 100 to output port 32.

4.2.4.3 WAIT

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

Version : Cassette, Disk

Format : WAIT port number , I[, J]

Remarks : port number is the port number, in the range 0 to 65535. I, J are integer expressions in the range 0 to 255.

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

Caution: It is possible to enter an infinite loop with the WAIT statement. If so, the machine needs to be restarted manually.

Example :

```
WAIT 32, 2
```

To suspend program execution until port 32 receives a 1 bit in the second bit position.

APPENDIX A

ERROR MESSAGE

Whenever BASIC detects an error, execution in direct or indirect mode will be suspended. An error message is displayed. It is possible to trap and test errors in a BASIC program using the ON ERROR GOTO statement and the ERL and ERR variables.

Apart from those listed in the below table, BASIC allows users to specify an error by use of the ERROR statement. Such error should be encoded a value of 0 through 255, preferrably 61 through 255.

All the BASIC messages with their associated code and number are listed below:

CODE	NUMBER	MESSAGE
NF	1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed unmatched FOR statement variable.
SN	2	Syntax error A line is encountered contains some incorrect sequence of characters (such as unmatched parentheses, misspelled command or statement, incorrect punctuation, etc.). Microsoft BASIC automatically enters edit mode at the line that carried the error.
RG	3	RETURN without GOSUB A RETURN statement is encountered for which there is no previous unmatched GOSUB statement.

OD	4	Out of DATA A READ statement is executed when there are no DATA statement with unread data remaining in the program.
FC	5	Illegal function call A parameter that is out of the range is passed to a math or string function. An FC error may also occur as the result of: 1. A negative or unreasonably large subscript. 2. A negative or zero argument with LOG. 3. A negative argument to SQR. 4. A negative mantissa with a noninteger exponent. 5. A call to an USR function for which the starting address has not yet been given. 6. OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR or ON...GOTO.
OV	6	Overflow The result of a calculation is too large to be represented in BASIC's numberr format. If underflow occurs, the result is zero and execution continues without an error.
OM	7	Out of memory A program is too large, has too many files, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.
UL	8	Undefined line number A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.

BS	9	Subscript out of range An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.
DD	10	Redimensioned array Two DIM statements are given for the same array, or DIM statement is given for an array after the default dimension of 10 has been established for that array.
/0	11	Division by zero A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. It is not necessary to fix this condition, because the program continues running. Machine infinity with the sign of the number being divided is the result of the division; or positive machine infinity is the result of the exponentiation.
ID	12	Illegal direct A statement that is illegal in direct mode is entered as a direct mode command.
TM	13	Type mismatch A string variable name is assigned a numeric value or vice versa; a function that excepts a numeric argument is given a string argument or vice versa.
OS	14	Out of string space String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.

LS	15	String too long An attempt is made to create a string more than 255 characters long.
ST	16	String formula too complex A string expression is too long or too complex. The expression should be broken into smaller expressions.
CN	17	Can't continue An attempt is made to continue a program that: 1. has halted due to an error 2. has been modified during a break in execution, or 3. does not exist.
UF	18	Undefined user function FN function is called before defining it with the DEF FN statement.
	19	Device I/O error An I/O error occurred on a cassette, disk, printer or CRT operation. It is a fatal error; i.e., BASIC cannot recover from the error.
	20	Verify error The current program is different from the program saved on that cassette.
	21	No RESUME An error trapping routine is entered but contains no RESUME statement.
	22	RESUME wihout error A RESUME statement is encountered before an error trapping routine is entered.
	23	Unprintable error An error message is not available for the error condition which exists.

	This is usually caused by an ERROR with an undefined error code.
24	Missing operand An expression contains an operator with no operand following it.
25	Line buffer overflow An entered line has too many characters.
26	Unprintable errors These codes have no definitions. Should be 49 reserved for future expansion in BASIC.
50	FIELD overflow A FIELD statement attempts to allocate more bytes than were specified for the record length of a random file in the OPEN statement. Or, the end of the FIELD buffer is encountered while doing sequential I/O (PRINT#, INPUT#) to a random file.
51	Internal error An internal malfunction has occured. Report to Microsoft the conditions under which the message appeared.
52	Bad file number A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified by MAXFILES statement.
53	File not found A LOAD, KILL, or OPEN statement references a file that does not exists in the memory.

54	File already open A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.
55	Input past end An INPUT statement is executed after all the data in the file has been INPUT, or for null (empty) file. To avoid this error, use the EOF function to detect the end of file.
56	Bad file name All illegal from is used for the file name with LOAD, SAVE, KILL, NAME, etc.
57	Direct statement in file A direct statement is encountered while LOADing an ASCII format file. The LOADing is terminated.
58	Sequential after PUT Parameter after PUT command is input wrongly.
59	Sequential I/O only A statement to random access is issued for a sequential file.
60	File not OPEN The file specified in a PRINT#, INPUT#, etc. hasn't been OPENed.
61 . . 255	Unprintable error These codes have no definitions. Users may place their own error code definitions at the high end of this range.

APPENDIX B

DISK BASIC

I. TECHNICAL INFORMATION

For each disk drive that is mounted, the following information is kept in memory:

A. Drive Information

1. Attributes	Drive attributes are read from the information sector when the drive is mounted, and may be changed with the SET statement. Current attributes may be examined with the ATTR$ function.
2. Track Number	This is the current track while the disk is mounted. Otherwise, track number contains 255 as a flag that the disk is not mounted.
3. Modification Counter	This counter is incremented whenever an entry in the File Allocation Table is changed. After a given number of changes has been made, the File Allocation Table is written to disk.
4. Number of Free Clusters	This is calculated when the drive is mounted, and updated whenever a file is deleted or a cluster is allocated.

5. File Allocation Table

The File Allocation Table has a one-byte entry for every cluster allocated on the disk. If the cluster is free, this entry is 255. If the cluster is the last one of the file, this entry is 300 (octal) plus the number of sectors that were used from this cluster. Otherwise, the entry is a pointer to the next cluster of the file. The File Allocation Table is read into memory when the drive is mounted, and updated:

* When a file is deleted
* When a file is closed
* When modifications to the table total twice the number of sectors in a cluster (this can be changed in custom verions)
* When modifications to the table have been made and the disk head is on (or passes) the directory track

B. Directory Format

On the diskette, each sector of the directory track contains eight file entries. Each file entry is 16 bytes long and formatted as follows:

Bytes	**Usage**
0-8	Filename, 1 to 9 characters. The first character may not be 0 to 255.

9	Attributed: &O 200 Binary file &O 10 force read after write check &O 20 write protected file Excluding &O 200, these bits are the same for the disk attributed byte which is the first byte of the information sector.
10	Pointer into File Allocation Table to the first cluster of the file's cluster chain.
11-15	Reserved for future expansion.

If the first byte of a filename is zero, that file entry slot is free. If the first byte is 255, that slot is the last occupied slot in the directory, i.e. this flags the end of the directory.

C. File Block

Each file on the disk has a file block that contains the following information:

1. File Mode (byte 0)

   This is the first byte (byte 0) of the file block, and its location may be read with VARPTR (#filenumber). The location of any other byte in the file block is relative to the file mode byte. The file mode byte is one of the following:

&01	Input only
&02	Output only
&04	File mode
&010	Append mode
&020	Delete file
&0200	Binary save

NOTE: It is not recommended that the user attempts to modify the next four bytes of the File Allocation Table. Many unforeseen complications may result.

2. Pointer to the File Allocation Table entry for the first cluster allocated to the file (+1)

3. Pointer to the File Allocation Table entry last cluster accessed (+2)

4. Last sector accessed (+3)

5. Disk number of file (+4)

6. The size of the last buffer read (+5). This is 128 unless the last sector of the file is not full (i.e., CTRL-Z)

7. This current position in the buffer (+6). This is the offset within the buffer for the next print or input.

8. File flag (+7), is one of the following:

&0100	Read after write check
&020	File write protected
&010	Buffer changed by PRINT
&04	PUT has been done. PRINT/INPUT have errors until a GET is done &02 Flag buffer is empty

9. Terminal position for TAB function and comma in PRINT statement (+8).

10. Beginning of sector buffer (+9), 128 bytes in length.

D. Disk Allocation

With Disk BASIC, storage space on the diskette is allocated beginning with cluster closest to the current position of the head. This method is optimized for writing. Custom versions can be optimized for reading. Disk allocation information is placed in memory when the disk is mounted and is periodically written back to the disk. Because this allocation information is kept in memory, there is no need of index blocks for random files, and there is no need to distinguish between random and sequential files.

E. Filename

A file is a collection of information, kept somewhere other than inside the computer's memory area, that stores programs.

There are two different ways to distinguish files, break them into categories and label them properly. One is called a "filename" and the other is called a "filenumber". These have been discussed in section 3.13.1.

The format for disk filename is:
drive # : filename . extension

F. File Format

Each file requires 137 bytes: 9 bytes plus the 128-byte buffer. Because the File Allocation Table keeps random access information for all files, random and sequential files are identical on the disk. The only distinction is that sequential files have a CTRL-Z (&O32) as the last character of the last sector. When this sector is read, it is scanned from the end for a non-zero byte. If this byte is CTRL-Z, the size of the buffer is set so that a PRINT overwrites this byte. If the byte is not CTRL-Z, the size is set so the last null seen is overwritten.

Any sequential file can be copied in random mode and remain identical. If a file is written to disk in random mode (i.e., with PUT instead of PRINT) and then read in sequential mode, it will still have proper end of file direction.

G. FORMATTING A DISK

In theory, the 5.25" floppy disks that you have purchased from your computer dealer are manufactured so that they can be used with any microcomputer. However, each microcomputer manufacturer designs his own method or format to store information (data) on a disk. That is why a program written for one machine is not necessarily usable on another machine.

Spectravideo requires a disk to undergo a process called formatting to prepare the disk to accept information sent from the computer to the disk drive.

Disk formatting is achieved by running two programs resided in the SV Extended BASIC Diskette, namely "svfrmt" and "format". If you want to create a diskette which can be booted automatically, the "sysgen.bas" program should be run as well.

The "svfrmt" utility program, which runs under the CP/M operating system, allows you to format a blank diskette for use either a CP/M diskette or a SV Extend Disk BASIC Diskette. The process will only be completed after the other program "format" is run.

The "format" program allows you to format a previously prepared diskette (using "svfrmt") for use as a SV Extended Disk BASIC Diskette.

Before mounting a drive with a new diskette, run BASIC's "format" program to initialize the directory (setting all bytes to 255), set the information sector to 0, and set all the File Allocation Table entries (except the directory track entry (254) to "free"(255).

II. COMMAND AND STATEMENT

The command and statements for BASIC program files are listed below. Most of these commands are described in Chapter 4.

ATTR$ (< drive > [#] filenumber , filename)
This returns a string of the current attributes for a drive, currently open file, or file that need not be opened.

CVI(< 2-byte string >)
CVS(< 4-byte string >)
CVD(< 8-byte string >)
Numeric values which are read in from a random disk file are converted from string to a figure. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. While CVD converts an 8-byte string to a double precision number.

DSKI$ (<drive> , <track> , <sector>)
This is the complementary function to the DSKO$ statement. DSSKI$ returns the contents of a sector (the first 255 bytes) to a string variable name.

DSKO$ (<drive> , <track> , <sector> , <string exp>)
This statement writes the string on the specified sector. The maximum length for the string is 256 characters. A string of fewer than 256 characters is zero-filled to the end.

EOF (<filenumber>)
Use to test for end-of-file while INPUTing. Otherwise an "Input past end" error message will be echoed. Return -1 (true) if the end of a sequential file has been reached.

[L] FILES [<drive number>]
Display the names of the files residing on a diskette. In addition to the filename the size of each file, in cluster, is output. (A cluster is the minimum unit of allocation for a file, being one half of a track.)

If <drive number> is omitted, the names of files on a diskette in drive 1 are listed. The command FILES2 lists those on the diskette in drive 2. LFILES outputs to the line printer.

Filenames of files created with OPEN or ASCII SAVE are listed with a space between its name and extension. Filename of binary files created with binary SAVE are listed with a decimal point between its name and extension. Files created by the SAVE <filename> command to save the current screen image are listed with a pound sign (#) between the name and the extension.

FPOS (<filenumber>)
FPOS returns the number of the physical sector where filenumber is located.

GET [#] <filename>
[, <record number>]
Read a record from a random disk file into a random buffer. If record number is omitted the next record after the last GET is read into the buffer. The largest possible record number is 32767. After a GET statement, INPUT# and LINE INPUT# may be done to read characters from random file buffer.

INPUT#
If the buffer is empty, write it; if the "buffer changed" flag is set, then read the next buffer.

IPL "RUN" + CHR$(34) + "1:
<filename> "
The IPL command instructs Disk BASIC to immediately execute the program you select when the Disk BASIC Diskette is booted.

KILL " <device descriptor>
<filename> "
Delete a file from the disk. This may be program file, sequential file or random-access data file.

LOAD " <device descriptor>
<filename> " [,R]
Load the specified program from the diskette into the computer's memory, and delete the current contents of memory. The option "R" permits you to run the program immediately after it is loaded that is equivalent to RUN. Also open data files are kept open.

LOC (<filenumber>)
With random disk files, LOC returns the record number just read or written from a GET or PUT. If the file was opened but no disk I/O has been performed yet, LOC returns a "0". With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was OPENed.

LSET <string variable> = <string exp>
RSET <string variable> = <string exp>
Move data from memory to a random file buffer, in preparation for a PUT statement. If <string exp> requires fewer bytes than were FIELDed to <string variable> , LSET left-justifies the string in the field and RSET right-justifies the string. Spaces are used to pad the extra position. If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field.

MAXFILES = <number of files>
To specify the maximum number of files opened at a time.

MERGE" <device descriptor> <filename> "
Load the program from diskette into memory, but does not delete the current contents of memory. The program line numbers on diskette are merged with the line numbers in

memory. If two lines have the same number, only the line from the diskette program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level. The MERGE command only merges files previously saved with the "A" option (ASCII files only). It does not merge machine code files or compressed binary format files.

MKI$ (< integer exp >)
MKS$ (< integer exp >)
MKD$ (< integer exp >)
Any numeric value that is placed in a random file buffer with a LSET or RSET statement is converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

NAME " <device descriptor>
<filename> " AS "
<device descriptor>
<filename> "
Rename a diskette file which may be program files, random files or sequential files. Only the filename is changed, the file is not modified and it remains in the same space and position on the diskette.

OPEN < filename> [FOR < mode>] AS [#] < filenumber>
To prepare a device for I/O operations in a file structure mode; where < mode> is one of the followings: INPUT, OUTPUT, APPEND.

The mode determines only the initial positioning within the file and the actions to be taken if the file does not exist. The action taken in each mode is:

INPUT The initial position is at the start of the file. An error is returned if the file is not found.

OUTPUT The initial position is at the start of the file. A new file is always created.

APPEND The initial position is at the end of the file. An error is returned if the file is not found.

If the FOR <mode> clause is omitted, the initial position is at the start of the file. If the file is not found, it is created. All variable records are 128 bytes in length.

When a file is OPENed for APPEND, the file mode is set to APPEND and the record number is set to the last record of the file. The program may subsequently execute disk I/O statements that move the pointer elsewhere in the file. When the last record is read, the file mode is reset to FILE and the pointer is left at the end of the file. Then, if you wish to append another record, execute GET# <n> , LOF(<n>)

This positions the pointer at the end of the file in preparation for appending.

At any one time, it is possible to have a particular filename OPEN under more than one filenumber. This allows different attributes to be used for different purposes. Or, for program clarity, you may wish to use different filenumbers for different methods of access. Each filenumber has a different buffer, so changes made under one file are not accessible to (or affected by) the other numbers until the record is written, e.g., GET# < n > , LOC(< n >).

PRINT#
Set the "buffer changed" flag. If the buffer is full, write it to disk. Then, if the end of file has not been reached, read the next buffer.

PUT [#] < filenumber >
[, < record number >]
To write a record from a random buffer to a random disk file. If < record number > is omitted the record will have the next available record number after the last PUT. The largest possible record number is 32767 while the smallest is 1.

PRINT# and PRINT# USING may be used to put characters in the random file buffer before a PUT statement. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

RUN " < device descriptor >
< filename > "
Load the program from diskette into memory and run it. This command deletes the contents of memory and closes all files before loading the program.

SAVE " < device descriptor >
< filename > "[,A]
Write the program to diskette that is currently residing in memory. Option "A" writes the program as a series of ASCII characters. Otherwise BASIC uses a compressed binary format. The "A" option requires a great deal more diskette storage space. It is mainly used for merging programs and transmitting files from one computer to another via a communication link.

SET <drive> [,[#] <filenumber>]
[< filename>], < attribute string>
The SET statement determines the attributes of the currently mounted disk drive, a currently open file or a file that need not be opened.

An attribute string is a string of characters that determines what attributes are set. It is confined to one of the followings:

R	Read after write
P	Write protect

Attributes are assigned in the following order:

1. **SET <drive> ,**
   <attribute string >
   This statement sets the current attributes for the disk. The attributes are permanently recorded.

2. When a file is created, the permanent file attributes recorded on the disk will be the same as the current drive attribute.

3. **SET <filename> ,**
   <attribute string>
   This statement changes the permanent file attributes that are stored in the directory entry for that file. It does not affect the drive attributes.

4. When an existing file is OPENed, the attributes of the filenumber are those of the directory entry.

5. **SET# <filenumber> ,**
   <attribute string>
   This statement changes the attributes for that filenumber but does not change the directory entry.

III. SEQUENTIAL DATA FILES

There are two different types of diskette data files that may be created and used by a BASIC program. One is a "sequential file" and the second is a "random access file".

Sequential files are easier to create than random files, but are limited in speed and flexibility when it comes to accessing data. The data is written sequentially, that is one item after the other, in the order it is sent to the diskette. It is loaded back into the computer in the same way.

The following steps must be included in a program to create and access a sequential file.

1. OPEN the file for output (from the computer to the disk drive) or appending (adding to it).

2. WRITE data to the file using the PRINT# command (or other commands).

3. CLOSE the file after you have written to it. To read data from a file you must OPEN it again for input (from the disk drive into the computer).

DEMO#1

The first demonstration program highlights the following four fundamental commands:

```
OPEN
CLOSE
PRINT
INPUT
```

```
10 OPEN "1"DEMO1" FOR OUTPUT AS #1
20 A = 10: B = 20
30 C = 30
40 PRINT#1,A;B;C
50 CLOSE#1
60 OPEN "1:DEMO1" FOR INPUT AS #1
70 INPUT#1,A,B,C
80 PRINT A,B,C
90 CLOSE#1
```

This program will save the numbers 10, 20 and 30 on the disk then read them and print them on the screen. Here's why:

Line 10 instructs the computer to OPEN (create) a file on drive 1 called DEMO#1 to which we will output, or write information. The #1 at the end of line 10 is the filenumber for the demo#1 file.

If you wish to open more than one file at a time, you must specify in your program how many files you wish to open. To specify the maximum number of files you will open at once, use the MAXFILES command. For example:

```
MAXFILES = 2
```

Line 40 is the one that actually instructs the computer to write them on the disk, and line 50 closes the demo#1 file (filenumber 1).

On line 60 the computer is instructed to reopen the file to be able to read the information back into the computer. Notice that the filenumber again is #1.

Line 70 causes the computer to read the information back into the computer, and line 80 prints out the specified variables. Line 90 closes the demo#1 file.

DEMO#2

This program illustrates the LINE INPUT# command.

```
10 OPEN "1:DEMO2" FOR OUTPUT AS #1
20 A$ = "THIS IS A DEMONSTRATION"
30 B$ = "THIS IS PART OF IT TOO"
40 PRINT#1,A$,B$
50 CLOSE#1
60 OPEN"1:DEMO2" for input as #1
70 LINE INPUT#1, A$
80 CLOSE#1
```

This program writes the message contained on lines 20 and 30 on the disk, then reads it back and prints it on the screen. The new command line input# appears on line 70. This command reads an entire line (up to 254 characters), without delimeters, from a sequential file to a string variable.

DEMO#3

This program demonstrates how to append new information to an existing sequential file.

```
10  OPEN"1:DEMO 3" FOR OUTPUT AS #1
20  A$ = "THIS IS A DEMONSTRATION"
30  B$ = "THIS IS PART OF IT TOO"
40  PRINT#1, A$, B$
50  CLOSE#1
60  C$ = "SO IS THIS"
70  OPEN"1:DEMO3" FOR APPEND AS #1
80  PRINT#1,C$
90  CLOSE#1
100 OPEN "1:DEMO3" FOR INPUT AS #1
110 LINE INPUT#1,D1$
120 LINE INPUT#1,C1$
130 PRINT D1$: PRINT C1$
140 CLOSE#1
```

Lines 10-50 are the same as those in the demo#2 program above. Lines 70-90 reopen the demo#3 file and write the message contained in C$. Then lines 100-140 open data file demo#3, then read in D1$ (which consist of A$ and B$) and C1$ (which consists of C$) and then print D1$ and C1$.

DEMO#4

This program demonstrates the last major command needed for sequential data file creation and access. The command is EOF, which is the abbreviation for "End of File".

```
10  OPEN"1:DEMO4" FOR OUTPUT AS #1
20  FOR A = 0 TO 50
30  PRINT#1,A
40  NEXT A
50  CLOSE#1
60  OPEN"1:DEMO4" FOR INPUT AS #1
70  IF EOF(1) THEN GOTO 120
80  INPUT#1,A
90  PRINT A
100 GOTO 70
110 CLOSE#1
120 PRINT"ALL DONE"
```

This program writes the numbers 0-50 into a file and then reads them back and prints them on the screen. It prints the message "ALL DONE" when it finishes. Delete line 70 from the program, change line 100 to read "GOTO 80" and then run the program.

The following error message will greet you:

```
INPUT PAST END IN 80
```

After the computer prints the last item in the filenumber 50-it returns to the file looking for more data to read because line 100 sent it to line 80 which tells it to read. But since there is no more data left in the file, you are told that you tried to input (transfer from disk to computer) past the end of the file.

The EOF function tests to see whether or not the end of a file is reached. If the end of a file has been reached (true) then the value that EOF returns (transmits) to the program is one (1). A zero (0) will be returned if the end of the file has not been reached.

This is what line 70 does: if the end of the file has been reached, then goto 120. Before each item is read, the EOF tests to see if the end of file has been reached. If it has not been reached (the false or zero condition), the program continues to line 80. However if the EOF test reports a true (1) condition then the program jumps to line 120 and prints the "ALL DONE" message rather than the "Input past end" error message.

IV. RANDOM ACCESS FILES

Creating and accessing random files requires more programming steps than is the case with sequential files. Random files are stored in the tokenized format while a sequential file is stored as ASCII characters.

The biggest advantage of random files is that data can be accessed anywhere on the diskette. This means that, unlike sequential files, it is not necessary to read through all the files one after another until the file you desired is found. This is so because the information that comprises a random file is stored and accessed in distinct units called "records", and each record is numbered.

The following programming steps are required to create a random file.

1. **OPEN** a file for random access.

2. The data must first be moved from the program area of memory to a random buffer prior to writing it on a disk. The **FIELD** command allocates space for the data in the random buffer.

3. Use the **LSET** OR **RSET** commands to position the data in the random buffer.

4. Write the data from the buffer to the diskette using the **PUT** statement. You need not close a random file before accessing (reading) the information back into the computer (as was the case with sequential files).

The following programming steps are required to access a random file.

1. **OPEN** a random file, if it was previously closed.

2. Use the **FIELD** statement to allocate space in the random buffer, if the file was previously closed.

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

DEMO#5

```
10  INPUT"CUSTOMER NAME:";Q$
20  INPUT"CITY:";R$
30  OPEN"1:DEMO5" AS #1
40  FIELD #1, 20 AS N$, 10 AS A$
50  LSET N$ = Q$
60  LSET A$ = R$
70  PUT#1,18
80  CLOSE#1
90  OPEN"1:DEMO5" AS #1
100 FIELD #1, 20 AS N$, 10 AS A$
110 GET #1,18
120 PRINT N$: PRINT A$
130 CLOSE#1
```

This program is the beginning of a database to hold customer names and their cities. It could be written as:

```
10  INPUT"CUSTOMER NAME:";Q$
20  INPUT"CITY:";R$
30  OPEN"1:DEMO5" AS #1
40  FIELD #1, 20 AS N$, 10 AS A$
50  LSET N$ = Q$
60  LSET A$ = R$
70  PUT#1,18
80  GET#1,18
90  PRINT N$: PRINT A$
100 CLOSE#1
```

Here is how the program works:

Lines 10 and 20 store the customer information in strings Q$ and R$.

Line 30 opens demo#5. Line 40 allocates the space for the information about the customers in a random buffer. It allocates 20 positions (bytes) for N$, and 10 positions for A$. N$ and A$ are the string variables in the string buffer that will hold the information about the customers that was originally in Q$ and R$.

The LSET commands in lines 50 and 60 move the data from the Q$ and R$ variables and places them into the string variables, N$ and A$ which are in the random buffer. Line 70 writes the record (the data) from the random buffer to the data file. The number 18 is the number of the record that we have arbitrarily chosen. You should be careful when you number your records because organization is the key to moving the data around among the program area, the random buffer and the random file. The GET command reads the data back into the random buffer from a random file.

The LSET command justifies the string variable to the left, and the RSET command justifies it to the right.

DEMO#6

Our previous program (demo#5) used only string variables. However, there will probably be many situations where you need to store numerical information in a random access file too. Before doing so, you must add on two extra programming steps. The first step coverts a numeric type value into a stirng type value before you write the data to the diskette. The second extra step converts the string variable type back into its numeric value. The following program demonstrates two of the commands that perform this conversion.

```
10  INPUT"CUSTOMER NAME:";CUST$
20  INPUT"CITY:";CITY$
30  INPUT"PHONE NUMBER:";TEL
40  TEL$ = MKD$(TEL)
50  OPEN "1:DEMO6" AS #1
60  FIELD#1, 20 AS N$, 10 AS A$, 8 AS T$
70  LSET N$ = CUST$
80  LSET A$ = CITY$
90  LSET T$ = TEL$
100  PUT#1, 18
110 GET#1, 18
120 T = CVD(T$)
130 PRINT N$: PRINT A$: PRINT T
```

This program writes the customer's name, city and telephone number on the disk, reads it back, and prints it on the screen. The new commands introduced in this program are on lines 80 and 110. Line 80 uses the MKD$ command to convert the numeric data stored in "tel" into a string variable called t$. This allows the telephone number to be written to the disk along with the other customer information which was typed in string form by the user. Later, after the information from the random file has been read, the CVD command converts the string varaible T$ into a numeric value which is stored in "T".

ADVANCED USES OF FILE BUFFERS

1. Information may be passed from one program to another by FIELDing it to an unopened file number (not #0). The FIELD buffer is not cleared as long as the file is not OPENed.

2. The FILEDed buffer for an unopened file can also be used to format strings. For example, an 80-character string could be placed into a FIELDed buffer with LSET. The strings could then be accessed as four 20-character strings using their FIELDed variable names. For example, instead of using the statement

```
   FIELD#1, 80 AS A$
```

   The alternative is

```
   FIELD#1, 20 AS A1$, 20 AS A2$,
   20 AS A3$, 20 AS A4$
```

3. FIELD#0 may be used as a temporary buffer, but note that this buffer is cleared after each of the following commands: FILES, LOAD, SAVE, MERGE, RUN, DSKI$, DSKO$, OPEN.

V DISK BASIC ERROR MESSAGE

Field overflow	A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
Internal error	An internal malfunction has occured in Disk BASIC. Report to Microsoft the conditions under which the message appeared.

Bad file number	A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.
File not found	A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.
File already open	A sequential output mode. OPEN is issued for a file that is already open; or a KILL is given for a file that is open.
Disk I/O error	An I/O error occurred on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.
File already exists	The filename specified in a NAME statement is identical to a filename already in use on the disk.
Disk full	A disk storage space is in use.
Input past end	An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.
Bad record number	In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to 0.
Direct statement in file	A direct statement is encountered while LOADing an ASCII-format file. The LOADing is terminated.
Too many file	An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

APPENDIX C

CONVERTING PROGRAMS TO SPECTRAVIDEO PERSONAL COMPUTER BASIC

Since SVI Personal Computer BASIC is very similar to many microcomputer BASIC's, the SVI Personal Computer will support programs written for a wide variety of microcomputers. If you have program written in a BASIC other than SVI Personal Computer BASIC, some minor adjustments may be necessary before running them with SVI Personal Computer BASIC. Here are some specific things to look for when converting BASIC programs.

File I/O	In SVI Personal Computer BASIC, you read and write information to a file on diskette or cassette by opening the file to associate it with a particular filenumber; then using particular I/O statements which specify that filenumber. I/O to diskette and cassette files is implemented differently in other BASIC's. Refer to section 3.13.1 for details on data file and to section 4.2.1.33 for "OPEN" statement. Also, in SVI Personal Computer BASIC, random file records are automatically blocked as appropriate to fit as many records as possible in each sector.
Graphics	How you draw on the screen varies greatly between different BASIC's. Refer to the discussion of graphics in section 3.13.2 for specific information about SVI Personal Computer graphics.

Logical Operations

In SVI Personal Computer BASIC, logical operations (NOT, AND, OR, XOR, IMP, and EQV) are performed bit-by-bit on integer operands to produce an integer result. In some other BASICs, the operands are considered to be simply "true" (non-zero) or "false" (zero) values, and the result of the operation is either true or false. As an example of this difference, consider this small program:

```
10 A=9:  B=2
20 IF A AND B THEN PRINT "BOTH A
   AND B ARE TRUE"
```

This example in another BASIC will perform as follows: A is non-zero, so it is true; B is also non-zero, so it is also true; because both A and B are true, A AND B is true, so the program prints: "BOTH A AND B ARE TRUE".

However, SVI Personal Computer BASIC calculates it differently: A is 1001 in binary form, and B is 0010 in binary form, so A AND B (calculated bit-by-bit) is 0000, or zero; zero indicate false, so the message is not printed, and the program continues with the next line.

This can affect not only tests made in IF statements, but calculations as well. To get similar results, recode logical expressions like the following:

```
10 A=9:  B=2
20 IF (A< >0) AND (B< >0)
   THEN PRINT "BOTH A AND B ARE
   TRUE"
```

IF...THEN

The IF statement in SVI Personal Computer BASIC contains an optional ELSE clause, which is performed when the expression being tested is false. Some other BASICs do not have this capability. For example, in another BASIC you may have:

```
10 IF A=B then 30
20 PRINT "NOT EQUAL"  :  GOTO 40
30 PRINT "EQUAL"
40 REM CONTINUE
```

This sequence of code will still function correctly in SVI Personal Computer BASIC, but it may also be conveniently recoded as:

```
10 IF A=B THEN PRINT "EQUAL" ELSE
   PRINT "NOT EQUAL"
20 REM CONTINUE
```

SVI Personal Computer BASIC also allows multiple statements in both the THEN and ELSE clauses. This may cause a program written in another BASIC to perform differently. For example:

```
10 IF A=B THEN GOTO 100  :  PRINT
   "NOT EQUAL"
20 REM CONTINUE
```

In some other BASICs, if the test A=B is false, control branches to the next statement; that is, if A is not equal to B, "NOT EQUAL" is printed. In SVI Personal Computer BASIC, both GOTO 100 and PRINT "NOT EQUAL" are considered to be part of the THEN clause of the IF statement. If the test is false, control continues with the next program line; that is, to line 20 in this example. PRINT "NOT EQUAL" can never be executed.

This example can be recoded in SVI Persoanl Computer BASIC as:

```
10 IF A=B THEN 100 ELSE PRINT "NOT
   EQUAL"
20 REM CONTINUE
```

MAT Functions

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

Multiple Assignments

Some BASIC's allow statements of the form:

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

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

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

Multiple Statements

Some BASIC's use a backslash ( ) to seperate multiple statements on a line. With SVI Personal Computer BASIC, be sure all statements on a line are seperated by a colon(:).

PEEKs and POKEs

Many PEEKs and POKEs are dependent on the particular computer you are using. You should examine the purpose of the PEEKs and POKEs in a program in another BASIC, and translate the statement so it performs the same function on the SVI Personal Computer.

String Handling

String Length: Since strings in SVI Personal Computer BASIC are all variable length, you should delete all statements that are used to declare the length of strings. A statement such as DIM A$(I, J), which dimensions a string array for J elements of length I, should be converted to the SVI Personal Computer BASIC statement DIM A$(J).

Concatenation: Some BASIC's use a ccmma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for SVI Personal Computer BASIC string concatenation.

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

Other BASIC	**SVI Personal Computer BASIC**
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

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

Other BASIC	**SVI Perosnal Computer BASIC**
A$(I)=X$	MID$(A$,I,1)=X$
A$(I,J)=X$	MID$(A$,I,J-I+1)=X$

Relational Expressions

In SVI Personal Computer BASIC, the value returned by a relational expression, such as A   B, is either -1, indicating the relation is true, or 0, indicating the relation is false. Some other BASICs return +1 to indicate true. If you use the value of a relational expression in an arithmetic calculation, the results are likely to be different from what you want.

Remarks

Some BASIC's allow you to add remarks to the end of a line using the exclamation point (!). Be sure to change this to a single quote (') when converting to SVI Personal Computer BASIC.

Rounding of Numbers

SVI Personal Computer BASIC rounds single- or double-precision numbers when it requires an integer value. Many other BASIC's truncate instead. This can change the way your program runs, because it affects not only assignment statements (for example, I%=2.5 results in I% equal to 3), but also affects function and statement evaluations (for example, TAB(4.5) goes to the fifth position, A(1.5) is the same as A(2), and X=11.5 MOD 4 will result in a value of 3 for X): Note in particular that rounding may cause SVI Personal Computer BASIC to select a different element from an array than another BASIC - possibly one that is out of range!

Sounding the Bell

Some BASICs require PRINT CHR$(7) to send an ASCII bell character. In SVI Personal Computer BASIC, you may replace this statement with BEEP, although it is not required.

Other

The BASIC language on another computer may be different from the SVI Personal Computer BASIC in ways other than those listed here. You should become familiar with SVI Personal Computer BASIC as much as possible in order to be able to appropriately convert any function you may require.

APPENDIX D

MATHEMATICAL FUNCTIONS

Functions that are not available in Microsoft BASIC can be derived by using the following formulae:

Function	**Equivalent**		
Logarithm to base B	LOGB(X)	=	LOG(X)/LOG(B)
Secant	SEC(X)	=	1/COS(X)
Cosecant	CSC(X)	=	1/SIN(X)
Cotangent	COT(X)	=	1/TAN(X)
Inverse sine	ARCSIN(X)	=	ATN(X/SQR(1-X*X)
Inverse cosine	ARCCOS(X)	=	1.5708-ATN(X/SQR (1-X*X)
Inverse secant	ARCSEC(X)	=	ATN(SQR(X*X-1)) +(SGN(X)-1)*1.5708
Inverse Cosecant	ARCCSC(X)	=	ATN(1/SQR(X*X-1)) +(SGN(X)-1)*1.5708
Inverse Contangent	ARCCOT(X)	=	1.5708-ATN(X)
Hyperbolic since	SINH(X)	=	(EXP(X)-EXP(-X))/2
Hyperbolic cosine	COSH(X)	=	(EXP(X)+EXP(-X))/2
Hyperbolic tangent	TANH(X)	=	(EXP(X)-EXP((-X))/ (EXP(X)+EXP(-X))
Hyperbolic secant	SECH(X)	=	2/(EXP(X)+EXP(-X))
Hyperbolic cosecant	CSCH(X)	=	2/(EXP(X)-EXP(-X))
Hyperbolic cotangent	COTH(X)	=	(EXP(X)+EXP(-X))/ ((EXP(X)-EXP(-X))
Inverse hyperbolic sine	ARCSINH(X)	=	LOG(X+SQR(X*X+1))
Inverse hyperbolic cosine	ARCCOSH(X)	=	LOG(X+SQR(X*X-1)
Inverse hyperbolic tangent	ARCTANH(X)	=	LOG((1+X)/(1-X))/2
Inverse hyperbolic secant	ARCSECH(X)	=	LOG((1+SQR (1-X*X))/X)
Inverse hyperbolic cosecant	ARCCSCH(X)	=	LOG((1+SGN(X)*SQR (1+X*X))/X)
Inverse hyperbolic cotangent	ARCCOTH(X)	=	LOG((X+1)/(X-1))/2

APPENDIX E

ASCII CHARACTER CODE

DECIMAL CODE	HEXADECIMAL CODE	DEFINITION
1	1	[CONTROL]+A
2	2	[CONTROL]+B
3	3	[CONTROL]+C
4	4	[CONTROL]+D
5	5	[CONTROL]+E
6	6	[CONTROL]+F
7	7	[CONTROL]+G
8	8	[CONTROL]+H
9	9	[CONTROL]+I
10	A	[CONTROL]+J
11	B	[CONTROL]+K
12	C	[CONTROL]+L
13	D	[CONTROL]+M
14	E	[CONTROL]+N
15	F	[CONTROL]+O
16	10	[CONTROL]+P
17	11	[CONTROL]+Q
18	12	[CONTROL]+R
19	13	[CONTROL]+S
20	14	[CONTROL]+T
21	15	[CONTROL]+U
22	16	[CONTROL]+V
23	17	[CONTROL]+W
24	18	[CONTROL]+X
25	19	[CONTROL]+Y
26	1A	[CONTROL]+Z
27	1B	ESCAPE
28	1C	CURSOR RIGHT
29	1D	CURSOR LEFT
30	1E	CURSOR UP
31	1F	CURSOR DOWN
32	20	
33	21	!
34	22	"
35	23	#
36	24	$

DECIMAL CODE	HEXADECIMAL CODE	DEFINITION
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L

DECIMAL CODE	HEXADECIMAL CODE	DEFINITION
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t

DECIMAL CODE	HEXADECIMAL CODE	DEFINITION
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	
124	7C	
125	7D	
126	7E	
127	7F	
128	80	
129	81	
130	82	
131	83	
132	84	
133	85	
134	86	
135	87	
136	88	
137	89	
138	8A	
139	8B	
140	8C	
141	8D	
142	8E	
143	8F	
144	90	
145	91	
146	92	
147	93	
148	94	
149	95	
150	96	
151	97	
152	98	
153	99	
154	9A	
155	9B	
156	9C	

DECIMAL CODE	HEXADECIMAL CODE	DEFINITION
157	9D	
158	9E	
159	9F	
160	A0	⊞
161	A1	⬒
162	A2	◰
163	A3	⊞
164	A4	◱
165	A5	◨
166	A6	▚
167	A7	◧
168	A8	▢
169	A9	▢
170	AA	▢
171	AB	▢
172	AC	☰
173	AD	◰
174	AE	▢
175	AF	▣
176	B0	◲
177	B1	◲
178	B2	⊞
179	B3	⬓
180	B4	▢
181	B5	◳
182	B6	⊞
183	B7	⊞
184	B8	◱
185	B9	◳
186	BA	◩
187	BB	◩
188	BC	⧄
189	BD	⊠
190	BE	⧅
191	BF	▣
192	C0	▚
193	C1	●
194	C2	◱
195	C3	◳
196	C4	◰

DECIMAL CODE	HEXADECIMAL CODE	DEFINITION
197	C5	▏
198	C6	♣
199	C7	◆
200	C8	│
201	C9	▒
202	CA	◤
203	CB	▁
204	CC	◢
205	CD	▌
206	CE	◞
207	CF	♠
208	D0	◥
209	D1	─
210	D2	▒
211	D3	▕
212	D4	
213	D5	
214	D6	
215	D7	

APPENDIX F

CONVERSION TABLE

Decimal	Binary	Hexadecimal	Octal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

APPENDIX G

TECHNICAL INFORMATION

I. MEMORY ORGANISATION

SVI-318 has 32K RAM: 16K user addressable RAM and 16K non-addressable video display RAM. The 16K user addressable RAM resides in Page 02 (Bank 0) from hexadecimal address C000 to FFFF.

SVI-328 has 80K RAM: 64K user addressable RAM and 16K non-addressable video display RAM. 32K user addressable RAM resides in Page 02 (Bank 0) from hexadecimal address 8000 to FFFF. Another 32K user addressable RAM resides in Page 21 (Bank 2) from hexadecimal address 0000 to 7FFF.

There is no command to disable the video RAM. Use either of the followings to increase the unable memory size in your program.

(A) FRE (0)

Free force a garbage collection before returning the number of free bytes.

(B) CLEAR N1, N2

To set all numeric variables to zero, all string variables to null, and to close all open files; optionally to set the end of memory and the amount of stack space.

N1 sets the amount of string space while N2 sets the end address of memory.

The memory management is bank and page selection by hardware control.

The Disk BASIC DOS does not have the same format structure as the CP/M Xerox 820.

SVI-318/SVI-328 System

```
DISK BASIC
  MBASIC INTERPRETER (4K)
  DISK BASIC DOS (16K)
```

```
CP/M OS
   CP/M 2.2 (20K to 24K)
```

II. DISK BASIC AND CP/M

The start-up memory is Bank 0 Page 2 (8000H to FFFFH).

In MBASIC ROM interpreter bootstrap, 2K to 4K is reserved for working area. Therefore, 12815 bytes for SVI-318 and 29199 bytes for SVI-328 are available to use MBASIC.

For Disk BASIC DOS bytes bootstrap, 8K to 12K is reserved for system control. Therefore, 4807 bytes for SVI-318 and 21191 bytes for SVI-328 are available to run Disk BASIC.

USER FREE MEMORY	SVI-318	SVI-328
ROM BOOTSTRAP	12815	29199
DISK BASIC DOS	4807	21191

There are two disk operating system:

(A) The Disk BASIC DOS requires 12K to 16K for DOS utility.

(B) The CP/M OS requires 20K to 24K for system control area and utility.

The 64K user memory for SVI-328 is available once CP/M bootstraps. Since Page 02 and Page 22 are reserved for Disk BASIC, approximate 32K (Page 02 from 8000H to FFFFH) is available when you use MBASIC.

In the SVI-318/SVI-328 single user system, the SWITCH command forces exchange Page 02 (Bank 0) and Page 22 (Bank 2). However it should power up with Disk BASIC DOS bootstrap.

If SVI-318 is used, run CP/M with the dip switches S1, S2 and S5 of 64K RAM cartridge switched on.

The SVI-803 and the SVI-807 RAM Expansion Cartridges are used to expand the user memory up to a full 160K Bytes.

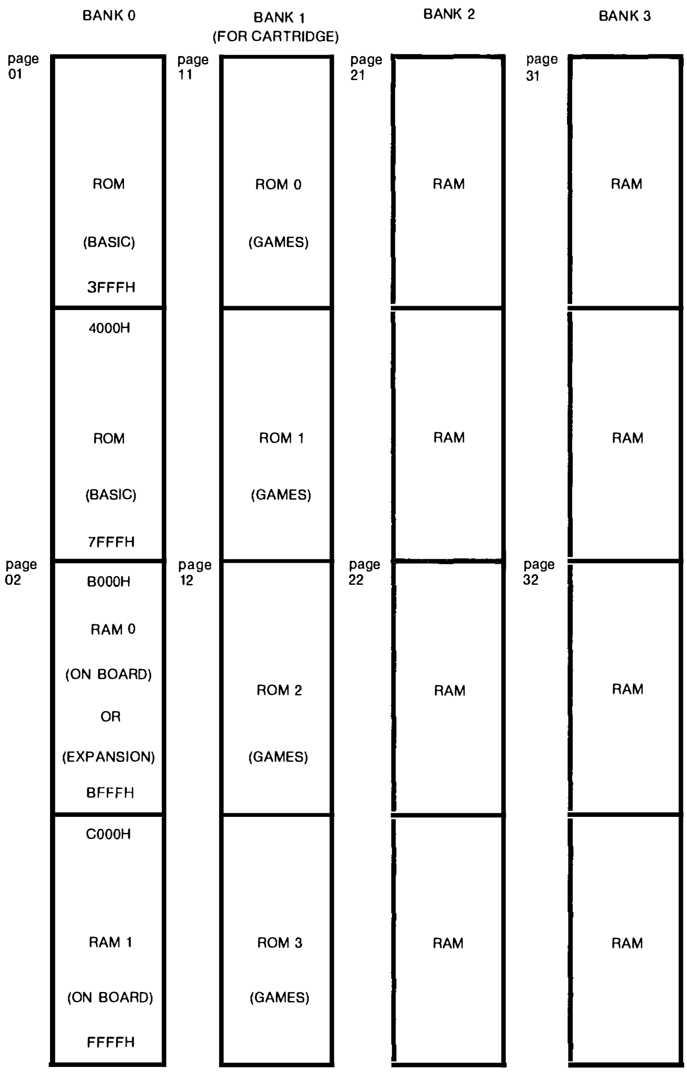
SVI-807 DIP SWITCH SELECTION

SVI-807		SVI-318	SVI-328
S1:	BK21	ON/OFF	OFF
S2:	BK22	ON/OFF	ON/OFF
S3:	BK31	ON/OFF	ON/OFF
S4:	BK32	ON/OFF	ON/OFF
S5:	BK02	ON*/OFF	OFF
S6:	48/32	OFF	OFF

Note: (1) Only two switches are allowed to switch on simultaneously.

(2) BK 02 can be selected only if 16K RAM Cartridge is not used.

Memory Map



Try the following memory bank RAM size test program:

```
10    REM  Memory bank RAM size test program
20    REM  Make sure you have 64K RAM Cartridge
30    REM  One have BK21 on, the other have BK31 on (in 32K option)
31    REM
40    CLEAR 10, &HD000
50    '
60    B2=&HD042:B3=&HD044
70    FOR K=&HD000 TO B3
80    READ A$:POKE K,VAL("&H"+A$)
90    NEXT K
100   DEF USR2=&HD011 'TEST BK 21
110   DEF USR3=&HD02B 'TEST BK 31
120   '---- INIT RAM AREA ----
130   FOR I=B2 TO B3+1
140   POKE I,0
150   NEXT I
160   A=USR3(0)
170   A=USR2(0)
180   PRINT"BANK 21 =";256*(PEEK(B2+1))+PEEK(B2)
190   PRINT"BANK 31 =";256*(PEEK(B3+1))+PEEK(B3)
200   STOP
210   REM --- DATA ----
220   DATA 21,00,00   :REM 'CHKSIZ:  LD HL,0 ;0-7FFFH
230   DATA 7E         :REM 'CHKSZ1:  LD A, (HL) ; READ
240   DATA 2F         :REM '         CPL
250   DATA 77         :REM '         LD (HL),A  ;WRITE
260   DATA BE         :REM '         CP (HL)
270   DATA 2F         :REM '         CPL
280   DATA 77         :REM '         LD (HL),A ;SAVE BACK
290   DATA C0         :REM '         RET NZ
300   DATA 23         :REM '         INC HL
310   DATA 70         :REM '         LD A,H     ;EXIT FOR HL=8000
320   DATA FE,80      :REM '         CP 80H
330   DATA 20,F3      :REM '         JR NZ,CHKSZ1
340   DATA C9         :REM '         RET        ;HL=SIZE
350   '
360   '  PSG,PORTB:  ROMEN1, ROMEN0,CAP,BK32,BK31,BK22,BK21,CART
370   '                  D7       D6    D5  D4   D3   D2   D1   D0
380   '   IN PSG DATA: 90H
390   '  OUT PSG DATA: 8CH
400   '  OUT PSG LATCH: 88H
410   '
420   DATA F3         :REM 'CHK21: DI
430   DATA 3E,0F      :REM '       LD A,0FH       :  PORT B
440   DATA D3,88      :REM '       OUT (88H),A    ;  LATCH
450   DATA DB,90      :REM '       IN A,(90H)     ;  CURRENT BANK COND
460   DATA 47         :REM '       LD B,A         ;  B=OLD BANK COND
470   DATA E6,FD      :REM '       AND 11111101B  ;  BANK 21 ON
480   DATA D3,8C      :REM '       OUT (8CH),A
490   DATA 21,00,00   :REM '       LD HL,0000     ;  no meaning
500   DATA CD,00,D0   :REM '       CALL CHKSIZ    ;  RESULT IN HL
510   DATA 22,42,D0   :REM '       LD (BK21),HL   ;  SAVE RAM SIZE
520   DATA 78         :REM '       LD A,B         ;  A = ORG BANK COND
530   DATA D3,8C      :REM '       OUT (8CH),A
540   DATA FB         :REM '       EI
550   DATA 09         :REM '       RET
570   DATA F3         :REM 'CHK31: DI
580   DATA 3E,0F      :REM '       LD   A,0FH     ;  PORT
590   DATA D3,88      :REM '       OUT (88H)      ;  LATCH
600   DATA DB.90      :REM '       IN A,(90H)
```

```
610  DATA 47          :REM '        LD B,A
620  DATA E6,F7       :REM '        AND 11110111B ;  BANK 31 ON
630  DATA D3,8C       :REM '        OUT (8CH),A   ;
640  DATA CD,00,D0    :REM '        CALL CHKSIZ   ;  RESULT IN HL
650  DATA 22,44,D0    :REM '        LD (BK31),HL  ;  DATA SAVE
660  DATA 78          :REM '        LD A,B        ;  ORG BANK COND
670  DATA D3,8C       :REM '        OUT (8CH),A
680  DATA FB          :REM '        EI
690  DATA C9          :REM '        RET
700  DATA 00,00       :REM '        BK21:   DS 2  ;  MEMORY SIZE OF BANK 21
710  DATA 00,00       :REM '        BK22:   DS 2  ;  MEMORY SIZE OF BANK 31
720  FOR I=&HD000 TO B3+1
730  LPRINT HEX$(I);" ";HEX$(PEEK(I))
740  NEXT
```

APPENDIX H

GLOSSARY

This part of the book explains many of the technical terms you may run across while programming in BASIC.

absolute coordinate form	In graphics, specifying the location of a point with respect to the origin of the coordinate system.
access time	The time between the instant, that an address is sent to a memory location and the instant data returns.
accumulator	One of several registers which temporarily store, or "accumulate" the results of various operations.
accuracy	The quality of being free from error. On a machine this is actually measured, and refers to the size of the error between the actual number and its value as stored in the machine.
adapter	A mechanism for attaching parts.
address	The location of a register, a particular part of memory, or some other data source or destination. Or, to refer to a device or a data item by its address.
addressable point	In computer graphics, any point in a display space that can be addressed. Such points are finite in number and form a discrete grid over the display space.

algorithm	A finite set of well-defined rules for the solution of a problem in a finite number of steps.
alphaumeric	Pertaining to a character set that contains letters and digits.
ALU	Arithmetic Logic Unit. The part of CPU that adds, subtracts, shifts, ANDs, ORs, and performs other computational and logical operations.
architecture	The organizational structure of a computer system.
array	A list of values stored in a series of memory locations.
ASCII	American Standard Code for Information Interchange. Consist of 128 letters, numbers, punctuation marks, and special symbols each of which consists of a binary pattern that uses eight digits.
assembler	A software program which converts symbolic or mnemonic language into machine language.
BASIC	Beginners All Purpose Symbolic Instruction Code. A high level programming language designed for the beginning programmer.
baud	A unit by which signal speeds are measured. In micro processing, the baud rate refers to the number of bits per second.
binary	Pertaining to a condition that has two possible values or states. Also, refers to the base 2 numbering system.

bit	A binary digit. Single element of a binary number with a value of either 0 or 1.
blank	A part of a data medium in which no characters are recorded. Also, the space character.
blinking	An international regular change in the intensity of a character on the screen.
boolean value	A numeric value that is interpreted as "true" (if it is not zero) or "false" (if it is zero).
bootstrap	A technique or device for loading the first instructions or words of a routine into memory. These instructions are used then to bring in the rest of the routine.
bps	Bit per second.
branch	A way of rerouting a program so that it branches to another set of instructions to perform another task.
bubble sort	A technique for sorting a list of items into sequence. Pairs of items are examined, and exchanged if they are out of sequence. This process is repeated until the list is sorted.
buffer	An area of storage which is used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transferring data from one device to another. Usually refer to an area reserved for I/O operations, into which data is read or from which data is written.

bug	An error in a program.
bus	A set of wires or conductors arranged in parallel, used to transmit data, signals, or power between parts of a computer system.
byte	The representation of a character in binary. Consist of eight bits.
clock	A device or circuit that sends out timing pulses to synchronize the action of the processor.
COBOL	Common Business Oriented Language. A high level language used in many business applications.
command	An instruction to the computer that causes something to happen.
compiler	A program to convert a high level language into assembly or machine language (understood by the computer).
concatenation	The operation that joins two strings together in the order specified, forming a single string with a length equal to the sum of the lengths of the two strings.
constant	A fixed value or data item.
control character	A character whose occurrence in a particular context initiates, modifies, or stops a control operation. A control operation is an action that affects the recording, processing, transmission, or interpretation of data; for example, pressing ENTER, font change, or end of transmission.

controller	An interface which allows the control of an I/O device by the CPU.
coordinates	Numbers which identify a location on the display.
CPU	Central Processing Unit. The part of the computer that controls all execution of instructions and arithmetic operations.
CRT	Cathode Ray Tube. The display on which information is shown after program execution.
cursor	A movable marker that is used to indicate a position on the display.
data	Essentially, information that is input to the computer.
data bus	An electrical path along which information passes.
debug	To find and eliminate mistakes in a program.
default	A value or option that is assumed when none is specified.
delimiter	A character that groups or separates words or values in a line of input.
diagnostic	Pertaining to the detection and isolation of a malfunction or mistake.
directory	A table of identifiers and references to the corresponding items of data. For example, the directory for a diskette contains the names of files on the diskette (identifiers), along with information that tells DOS where to find the file on the diskette.

disabled	A state that prevents the occurrence of certain types of interruptions.
disk	A plate resembling a record album with a magnetic surface used to store data or programs. Also known as "floppy disk".
DOS	Disk Operating System. In this book, refers only to the SVI Personal Computer Disk Operating System.
dummy	Having the appearance of a specified thing but not having the capacity to function as such. For example, a dummy argument to a function.
dump	The transfer of information from one piece of equipment to another.
duplex	In data communications, pertaining to a simultaneous two-way independent transmission in both directions. Same as full duplex.
dynamic	Occurring at the time of execution.
echo	To reflect received data to the sender. For example, key pressed on the keyboard is usually echoed as characters displayed on the screen.
edit	To enter, modify, or delete data.
editor	A program used for the creating and/or altering of text in another program.
element	A member of a set; in particular, an item in an array.
enabled	A state of the processing unit that allows certain types of interruptions.

end of file (EOF)	A "marker" immediately following the last record of a file, signalling the end of that file.
event	An occurrence or happening; in particular to the events tested by KEY(n), STRIG(n).
execute	To perform an instruction or a computer program.
expression	A particular grouping of numbers, letters, or variables that comprise a single quantity.
extent	A continuous space on a diskette, occupied or reserved for a particular file.
fault	An accidental condition that causes a device to fail to perform in a required manner.
fetch	Read out of an instruction/data from an addressed memory location.
field	In a record, a specific area used for a particular category of data.
file	A set of related records treated as a unit.
firmware	The programs that are built into the ROM of a microcomputer.
fixed-length	Referring to something in which the length does not change. For example, random files have fixed-length records; that is, each record has the same length as all the other records in the file.

flag	Any of various types of indicators used for identification, for example, a character that signals the occurrence of some condition.
floppy disk drive	A peripheral device used to store data and input data to the computer. It is also known as an input/output device.
flowchart	A diagram used in the development of a computer program. A flowchart shows the sequence of steps to be taken.
folding	A technique for converting data to a desired form when it doesn't start out in that form. For example, lowercase letters may be folded to uppercase.
font	A family or assortment of characters of a particular size and style.
foreground	The part of the display area that is the character itself.
format	The particular arrangement or layout of data on a data medium, such as the screen or a diskette.
form feed	A character that causes the print or display position to move to the next page.
FORTRAN	FORmula TRANslation. A high level language using algebraic notation.
function	A procedure which returns a value depending on the value of one or more independent variables in a specified way. More generally, the specific purpose of a thing, or its characteristic action.

function key	One of the ten keys labeled F1 through F10 on the top left side of the keyboard.
garbage collection	Synonym for housecleaning.
gate	An electrical signal circuit, with two (or more) inputs and one output, that behaves as a switch to create a particular state (either a binary one or zero).
graphic	A symbol produced by a process such as handwriting, printing, or drawing.
half duplex	In data communication, pertaining to an alternate, one way at a time, independent transmission.
hard copy	A printed copy of machine output in a visually readable form.
hardware	The physical components that make up a particular computer system, include all the peripheral devices.
hexadecimal	A numbering system uses the digits 0-9 and the letters A-F.
header record	A record containing common, constant, or identifying information for a group of records that follows.
hexadecimal	A numbering system uses the digits 0-9 and the letters A-F.
hertz (Hz)	A unit of frequency equal to one cycle per second.

hierarchy	A structure having several levels, arranged in a tree-like form. "Hierarchy of operations" refers to the relative priority assigned to the relative priority assigned to arithmetic or logical operations which must be performed.
high level language	A programming language that is easier to understand and more convenient for the programmer. BASIC, FORTRAN, PASCAL and PL-1 are some examples of high level languages.
host	The primary or controlling computer in a multiple computer installation.
housecleaning	When BASIC compresses string space by collecting all of its useful data and frees up unused areas of memory that were once used for strings.
I/O devices	Input/Output devices such as disk drive, data cassette, keyboard, printer, TV monitor, etc.
implicit declaration	The establishment of a dimension for an array without it having been explicity declared in a DIM statement.
increment	A value used to alter a counter.
initialize	To set counters, switches, addresses, or contents of memory to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine.
instruction	In a programming language, any meaningful expression that specifies one operation and its operands, if any.

instruction set	The set of instructions built into the firmware of the microcomputer. This instruction set is used by the programmer.
integer	One of the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
integrity	Preservation of data for its intended purpose; data integrity exists as long as accidental or malicious destruction, alteration, or loss of data are prevented.
interface	A shared boundary through which peripheral devices are linked to the mainframe console of the micro-computer.
interpret	To translate and execute each source language statement of a computer program before translating and executing the next statement.
interpreter	A program that converts one instruction at a time into machine language understood by the computer.
interrupt	To stop a process in such a way that it can be resumed.
invoke	To activate a procedure at one of its entry points.
joystick	A lever that can pivot in all directions and is used as a locator device.
justify	To align characters horizontally or vertically to fit the positioning constraints of a required format.
keyboard	This is the console of the computer in which data is input to the CPU.

keyword	One of the predefined words of a programming language; a reserved word.
kilobyte (K)	When referring to memory capacity, two to the tenth power or 1024 in decimal notation.
library	A collection of files or records that can be accessed easily.
line	When referring to text on a screen or printer, one or more characters output before an ENTER to the first print or display position. When referring to input, a string of characters accepted by the system as a single block of input; for example, all characters entered before you press the ENTER key. In graphics, a series of points drawn on the screen form a straight line. In data communications, any physical medium, such as a wire or microwave beam, that is used to transmit data.
line feed	A character that causes the print or display position to move to the corresponding position on the next line.
literal	An explicit representation of a value, especially a string value; a constant.
load	To enter a program into a computer's memory.
location	Any place in which data may be stored.
logic	A particular way of reasoning using thought processes.

loop	A set of instructions that may be executed repeatedly while a certain condition is true.
Mega (M)	One million. When referrring to memory, two to the twentieth power (1,048,576 in decimal notation).
machine infinity	The largest number that can be represented in a computer's internal format.
mantissa	For a number expressed in floating point notation, the numeral that is not the exponent.
mask	A pattern of characters that is used to control the retention or elimination of another pattern of characters.
matrix	An array with two or more dimensions.
matrix printer	A printer in which each character is represented by a pattern of dots.
memory	The part of the computer that stores data and instructions. Each instruction uses a particular address which tells the CPU where to fetch from.
menu	A list of available operations.
microprocessor	This is also known as the CPU. It comprises of one or more LSI circuits that control all the processes of the computer.
mini-floppy	A 5-1/4 inch diskette.
mnemonics	These are abbreviated terms for instructions, used so that the programmer can easily remember them.

modem	Modulator DEModulator. This is a device used to convert data to signals that can be transmitted over telephone lines and then back to data again at the receiving end.
motherboard	This is usually the main board on which all the components are seated.
multi-processing	The process of executing more than one program almost at the same time via multiprocessor and/or time robin.
nest	To incorporate a structure of some kind into another structure of the same kind. For example, you can nest loops within other loops, or call subroutines from other subroutines.
nibble	Half a byte; consisting of four bits.
notation	A set of symbols, and the rules for their use, for the representation of data.
nonvolatile storage	This is a mode of storage which, when the power is shut off, the stored data is still retained.
null	Empty, having no meaning. In particular, a string with no characters in it.
number-crunching	This is a way of describing a computer or a program that can handle large amounts of arithmetical operations.
object program	This is a program that has been translated into a machine language suitable for the computer.
octal	Pertaining to a base 8 numbering system.

offset	The number of units from a starting point (in a record, control block, or memory) to some other point. For example, in BASIC the actual address of a memory location is given as an offset in bytes from the location defined by the DEF SEG statement.
on condition	An occurrence that could cause a program interruption. It may be the detection of an unexpected error, or of an occurrence that is expected, but at an unpredictable time.
on-line	Whenever a peripheral device is interacting with its host computer, it is said to be "on-line". A printer is said to be "on-line" when it is doing a computer printout.
operand	That which is operated upon.
operation	A well defined action that, when applied to any permissible combination of known entities, produces a new entity.
operating system	Software that controls the execution of programs; often used to refer to DOS.
output	When data is said to be "output" it usually refers to the printout from a printer. Output may also be programs or data saved on a floppy diskette.
overflow	When the result of an operation exceeds the capacity of the intended unit of storage.
overlay	To use the same areas of memory for different parts of a computer program at different times.

overwrite	To record into an area of storage so as to destroy the data that was previously stored there.
pad	To fill a block with dummy data, usually zeros or blanks.
page	Part of the screen buffer that can be displayed and/or written on independently.
parameter	A name in a procedure that is used to refer to an argument passed to that procedure.
parity check	A technique for testing transmitted data. Typically, a binary digit is appended to a group of binary digits to make the sum of all the digits either always even (even parity) or always odd (odd parity).
peripheral	Any device external from the host computer but used in conjuction with the computer to perform operations such as printouts, data storage and retrieval, CRT displays, telecommunications, graphics, etc.
pixel	A graphic "point". Also, the bits which contain the information for that point.
pointer	This is the register in the CPU that contains the memory address.
port	An access point for data entry or exit.
position	In a string, each location that may be occupied by a character and that may be identified by a number.

precision	A measure of the ability to distinguish between nearly equal values.
program	The sequence of instructions that tell the computer what task to perform.
program counter	This is the register in the CPU that specifies the address of the next instruction to be executed.
prompt	A question the computer asks when it needs you to supply information.
protect	To restrict access to or use of all, or part of, a data processing system.
queue	A line or list of items waiting for service; the first item that went in the queue is the first item to be serviced.
random access memory (RAM)	Storage in which you can read and write to any desired location. Sometimes called direct access storage.
range	The set of values that a quantity or function may take.
raster scan	A technique of generating a display image by a line-by-line sweep across the entire display image by a line-by-line sweep across the entire display screen. This is the way pictures are created on a television screen. This is the way pictures are created on a television screen.
read-only	A type of access to data that allows it to be read but not modified.

read only memory (RAM)	This is the part of memory that may only be read from. It is said to be "nonvolatile" meaning that when power is removed the ROM retains its information.
record	A collection of related information, treated as a unit.
recursive	Pertaining to a process in which each step makes use of the results of earlier steps, such as when a function calls itself.
register	A circuit used to store or manipulate bits or bytes of data in the CPU.
relative coordinates	In graphics, values that identify the location of a point by specifying displacements from some other point.
reserved word	A word that is defined in BASIC for a special purpose, and that you cannot use as a variable name.
resolution	In computer graphics, a measure of the sharpness of an image, expressed as the number of lines per unit of length discernible in that area.
routine	Part of a program, or a sequence of instructions called by a program, that may have some general or frequent use.
row	A horizontal arrangement of characters or other expressions.
scale	To change the representation of a quantity, expressing it in other units, so that its range is brought within a specified range.

scan	To examine sequentially, part by part. See raster scan.
scroll	To move all or part of the display image vertically or horizontally so that new data appears at one edge as old data disappears at the opposite edge.
segment	A particular 64K-byte area of memory.
sequential access	An access mode in which records are retrieved in the same order in which they were written. Each successive access to the file refers to the next record in the file.
software	Software pertains to the programs that are input to the computer by the user.
source program	A program written in a language that is easily understood.
sprite	This is a shape designed by the programmer when using a computer's graphic capabilities.
stack	A method of temporarily storing data so that the last item stored is the first item to be processed.
statement	A meaningful expression that may describe or specify operations and is complete in the context of the BASIC programming language.
stop bit	A signal following a character or block that prepares the receiving device to receive the next character or block.
storage	A device, or part of a device, that can retain data. Memory.

string	A sequence of characters.
subroutine	A routine in a program that may be used over again to perform a specific function.
subscript	A number that identifies the position of an element in an array.
syntax	The rules governing the structure of a language.
table	An arrangement of data in rows and columns.
target	In an assignment statement, the variable whose value is being set.
Tele-communicatons	Synonym for data communication.
terminal	A device, usually equipped with a keyboard and display, capable of sending and receiving information.
time sharing	The process of sharing the use of a CPU via time robin for more than one user.
toggle	Pertaining to anything having two stable states; to switch back and forth between the two states.
trailing	Located at the end of a string or number. For example, the number 1000 has three trailing zeros.
trap	A set of conditions that describes an event to be intercepted and the action to be taken after the interception.
truncate	To remove the ending elements from a string.

truth table	A truth table shows the different values that an AND, OR, NAND, NOR or other logic gates will have, according to two select inputs.
two's complement	A form for representing negative numbers in the binary number system.
typematic key	A key that repeats as long as you hold it down.
update	To modify, usually a master file, with current information.
utility program	This is a program that helps the user perform various utility functions, such as a debugging program to find mistakes in programs.
variable	A quantity that can assume any of a given set of values.
variable-length record	A record having a lenth independent of the length of other records in the file.
vector	In graphics, a directed line segment. More generally, an ordered set of numbers, and so, a one-dimensional way.
wraparound	The technique for displaying items whose coordinates lie outside the display area.
write	To record data in a storage device or on a data medium.

APPENDIX I

INDEX

SVI™

SPECTRAVIDEO

US $19.95
UM-318/328-BRM.