Programmer Manual

SYKES Comm-Stor IV

87-02-04

Comm-Stor IV

PROGRAMMER MANUAL

P/N 9995ØBØ257B

2nd Edition


SYKES DATATRONICS, INC., Rochester, New York 14606
Sykes Publication No. 9990B0257B Printed in U.S.A.

Price of additional copies - $30.00

TABLE OF CONTENTS

This page intentionally blank.

CHAPTER 1

INTRODUCTION

CONTENTS

1. WELCOME TO Comm-Stor IV

The Comm-Stor® IV* system is a microprocessor-based communications storage unit which uses IBM compatible flexible diskettes (floppy disks) and an extended BASIC language package, referred to as Sykes BASIC**. (Any Commands not common to the computer industry will be described in detail.)

* Henceforth in this manual, Comm-Stor IV will be referred to as Comm-Stor.

**Sykes BASIC conforms with the NBS Minimal BASIC.

Comm-Stor has several new attributes which greatly increase programmer efficiency. These include:

a. **A File Management System** that reduces the maintenance of libraries and the addressing of files to an absolute minimum.

b. **A Real Time Control System** that allows the programmer to specify and control interrupts, based on time or changes (occurrences of change) in the device cables which interface with the unit.

c. **Virtual Arrays** used with Sykes BASIC that provide the programmer with greater flexibility when updating files and reduce the amount of internal memory needed to contain an array.

d. **Overlaying Programs** that allow different programs to be temporarily brought into memory and executed. This is ideal for the execution of subroutines, or when a program is too large to fit into the memory available.

Any type of peripheral device may be used with Comm-Stor. However, this manual refers to a standard ASCII terminal when describing the use of Comm-Stor.

2. PURPOSE

The purpose of this manual is to provide the programmer with all the information necessary for successful use of the options available in the Comm-Stor system. This manual is <u>not</u> intended to teach the BASIC language; it is assumed that the programmer has a fundamental knowledge of the BASIC language prior to reading this document. If the programmer is not familiar with the BASIC language, refer to the following book as a possible source of information:

BASIC, Albrecht, Finkel and Brown
John Wiley & Sons, New York, London, Sydney, Toronto
Sykes Publication Number 1030B5000

Additionally, the programmer is advised to study the Comm-Stor IV Reference Manual (Sykes Publication Number 9990B0-258A) before proceeding further in this manual. This is suggested as an aid in understanding the Comm-Stor Mode of operations.

Although Chapters 1 through 3 are directed toward the novice programmer with little experience, they also introduce the unique Sykes BASIC features. Chapters 4 through 6 are directed toward the more experienced programmer, or the programmer who has a confident knowledge of the BASIC language. Chapter 7 contains a brief description of all commands available in Sykes BASIC and Chapter 8 contains the available functions. The remaining chapters and appendices are designed as quick reference sources during operation of Comm-Stor.

3. POWER ON

After connecting Comm-Stor to an external power line, depress the POWER/CIRCUIT BREAKER switch on the rear panel. The RESTART button on the front panel will illuminate, indicating that power is on. If the lamp does not light, press the POWER/CIRCUIT BREAKER switch again. If the lamp still does not illuminate, service assistance should be sought. Power should also be applied to any attached peripheral devices. The order in which power is applied to various devices will not affect the operation of Comm-Stor.

At this point, Comm-Stor will display an identification message.

The system is now in the idle mode and ready to accept a diskette for data input or output.

4. DISKETTES

A. General

A flexible diskette is a thin disk permanently enclosed in a semi-rigid, protective, flexible jacket. The recording surface of the diskette is divided into "Tracks" and "Sectors", as illustrated in Fig. 1-1.

A "track" is a circular band of area that passes beneath the read/write head as the diskette revolves. A diskette has 77 tracks, the outermost being Track 0 and the innermost being Track 76. Each track is divided into 26 sectors, each of which contains 128 bytes. The sectors of each track are numbered from 1 to 26. Therefore, there are a total of 2002 (77 x 26) sectors on a diskette. Each sector may be separately addressed by a specified track and sector number. However, the powerful file management system residing in the Comm-Stor system keeps a record of the different files on the diskette; therefore, each file need only be referenced by name.

B. Selecting The Diskette

Configuration Diskette

The Configuration diskette is supplied by Sykes Datatronics, Inc. It is the device which allows the user to create User and Refresh diskettes, as well as vary the operations and resources of Comm-Stor. Use of the Configuration diskette is covered in Sykes Publication Number 9990B0259A, "Comm-Stor IV Configuration Manual."

Refresh Diskette

Once Comm-Stor is configured, it is possible to store this configuration on a diskette, called a Refresh diskette. After a Refresh diskette has been created, another Comm-Stor unit can be identically configured by inserting the Refresh diskette and pressing the RESTART button.

User Diskette

The User diskette contains a Directory and is used for all normal operations. If not already available, this diskette should be created before proceeding further in this manual.

The User diskette is created with the Configurator. It is initialized with such

Sykes DISKETTE

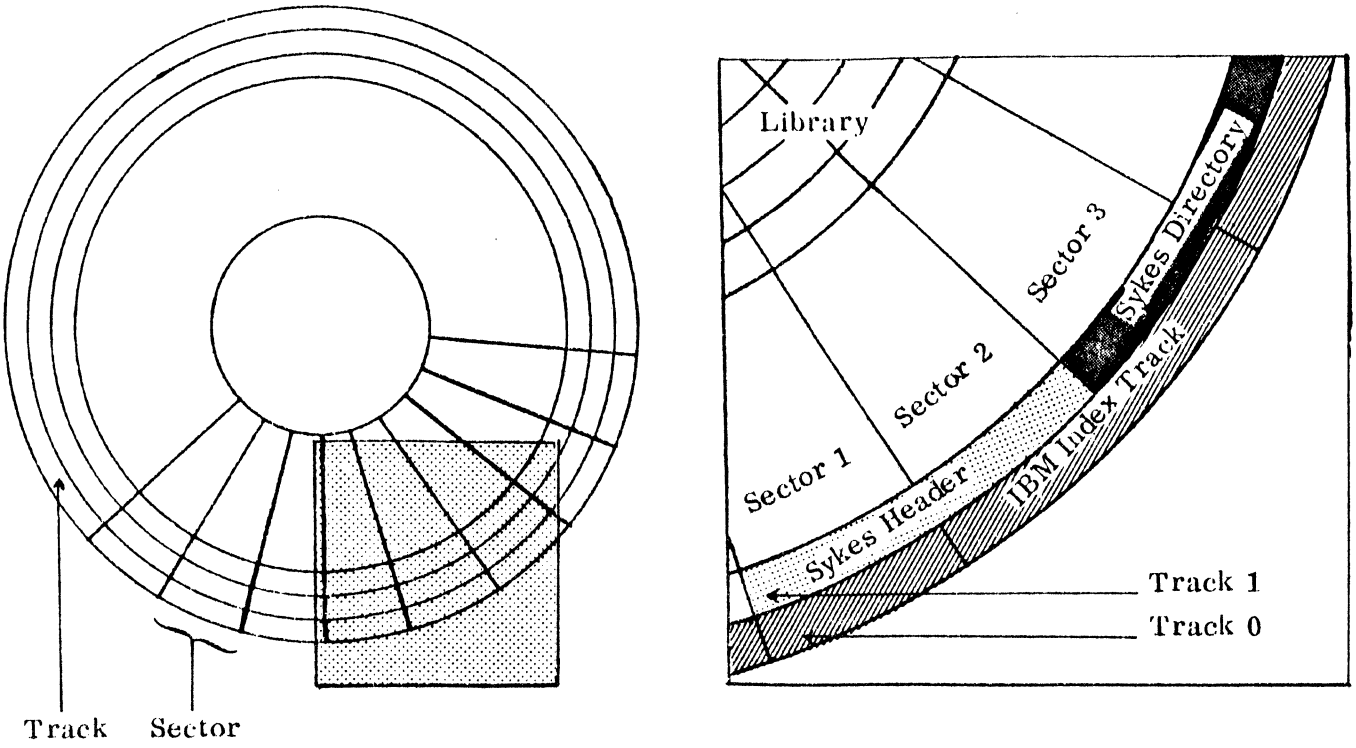


FIGURE #1-1

INSERTING THE DISKETTE

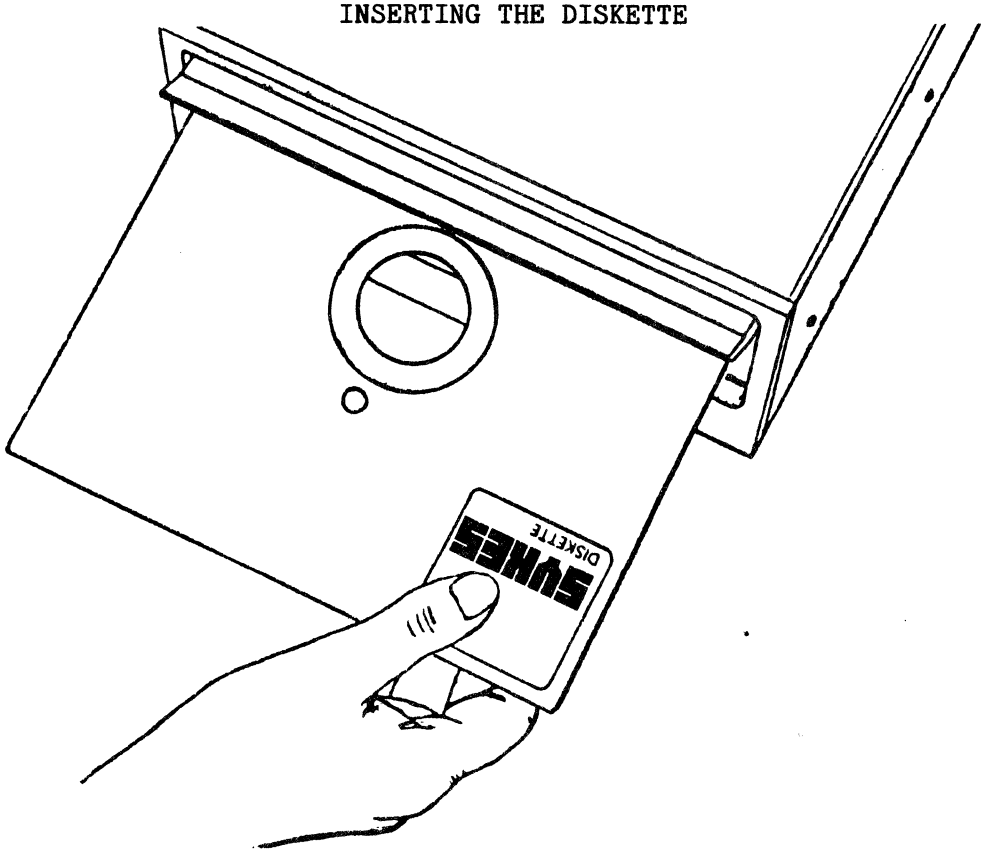


FIGURE #1-2

parameters as the maximum number of characters in the extension, and the maximum file length. A detailed description of User diskettes can be found in Appendix F.

C. **Inserting and Removing the Diskette**

The drive doors of all Comm-Stor units are equipped with an interlock which prevents them from closing unless a diskette is inserted.

To insert the diskette:

1.) Power Up the Comm-Stor Unit.

2.) Grasp the diskette between the thumb and index finger. The label should be face up and toward the operator (Fig. 1-2).

3.) Slide the diskette into the drive.

4.) Exert a slight inward pressure with the index finger and gently pull the drive door down with the thumb. DO NOT FORCE THE DOOR CLOSED!! A gentle horizontal pressure on the disk is sufficient to release the interlock. When the door closes, the READY light on the front panel will illuminate, indicating that the respective drive is ready for use.

IT IS IMPORTANT THAT DISKETTES BE INSERTED ONLY WITH POWER ON AS SPINDLE ROTATION ALIGNS THE DISKETTE. A DISKETTE NEED NOT BE REINSERTED IF POWER IS TURNED OFF AND ON AGAIN, PROVIDED THE DOOR OF THE DRIVE IS NOT DISTURBED.

To remove the diskette:

1.) Gently lift the drive door until it is fully open.

2.) Slide the diskette out of the drive.

D. **Diskette Care**

A diskette must be handled with care. Improper treatment or carelessness may result in the loss of data, and possibly, many hours of work. Observe the following precautions:

1. **Never touch the exposed diskette surface.** Handle the diskette only near the label.

2. **Do not write on the diskette cover;** write only on the label. If possible, write on the label before placing it on the diskette.

3. **Do not attempt to clean a dirty or dusty diskette;** such a diskette should be discarded.

4. **Keep the disk away from potentially magnetic materials (paper clips, etc.), or magnetic sources (unshielded power supplies, CRT monitors, etc.).** A magnetic field on such materials may result in the loss of data from the diskette.

5. **Do not bend the diskette.**

6. **Do not expose the diskette to extremes of heat or cold.**

7. **Keep the diskette in its protective cover when not in use.** Dust and liquid can damage the exposed diskette surface.

8. **Store diskettes vertically in boxes when not in use.**

5. DEFINITIONS

Some words contained in this manual may not be clear to the average programmer, or our intended usage may differ from that which is common to the industry. The following list defines some of these words as used in this manual.

Address - a number identifying a location where information is stored.

Array and Variable Buffer - that area in memory which contains all variable names, their characteristics, and their values (or addresses which direct the system to their values).

Average File Size - that result obtained in the Configurator by dividing the total amount of disk space by the maximum number of files specified by the user for a particular diskette.

BASIC Program Space - that area in the User Memory Space which consists of the File Buffer, the String Buffer, the Free Space Buffer, the Array and Variable Buffer, and the Program Text Buffer.

BASIC Work Space - that area of the User Memory Space which consists of the File Buffer, String Buffer, Free Space, and Array and Variable Buffer.

Bit - a binary digit.

Byte - a general term used to indicate a measurable portion of consecutive binary digits. Each byte contains 8 bits.

Byte Count - the number of bytes used to store data in a disk file.

Buffer - a portion of memory used to temporarily hold data input to or output from Comm-Stor.

Configuration Time - the time when the Configurator, residing on the Configuration diskette, is executed.

Constant - a number or a string of characters enclosed in quotes.

Device - a machine connected to Comm-Stor through one of the ports located on the back of the unit.

Disk File - a file which resides on a diskette.

Disk File Reference Number - (see file reference number).

Expression - any BASIC representation of a numeric or character value.

File - a collection of related records treated as a unit.

File Buffer - that area in memory used to temporarily hold data being transferred to and from disk files.

File Reference Number - a whole number between 4 and 24 (inclusive) which is associated with a disk file.

File Slot - a section on a User diskette which is the maximum size of the associated file name.

Fixed Space - those areas in the User Memory Space which are specified at configuration time (i.e., the Modem Buffer, the Terminal Buffer and the Object Buffer.)

Free Space Buffer - that area in memory which contains the total number of bytes currently unused by Comm-Stor.

Hard-copy - Paper containing printed characters, also known as a computer printout.

Invoker - the device (Terminal or Modem) that places Comm-Stor in the BASIC mode of operation.

Image - defines the format of data for input or output operations.

Library Slot - (see file slot).

Location - (see address).

Modem Buffer - that area in memory used to temporarily hold data sent from the device attached to the Modem Port to the Comm-Stor unit.

Object Buffer - that area in memory used to hold Sykes supplied programs.

Operator - special characters used to define the relation between values.

Port Reference Number - a whole number between 0 and 3 (inclusive) related to a device attached to Comm-Stor (see Reference Number).

Programmer - the person interacting directly with the Comm-Stor system.

Real Number - any whole number or a fraction of a whole number.

Real-Time Clock - an internal hardware register that interrupts the system at one second intervals allowing software to maintain the correct time of day.

Record - a collection of related data items treated as a unit.

Reference Number - a whole number (0-24) which is associated with a certain condition, item, or action as specified below:

0 - 3 device numbers, where
- 0 - invoker
- 1 - terminal
- 2 - modem
- 3 - printer

4 - 24 file reference numbers

Stack - that area in internal memory (located at the hexadecimal address "100") which temporarily holds intermediate values used by a BASIC program. It consists of 256 bytes and uses a maximum of 30 bytes for system overhead, about 20 bytes for FOR - NEXT loops, 6 bytes for any GOSUBs, 11 bytes for any CALLs, and 7 bytes for intermediate calculation values. The stack is constantly updated (i.e., when a GOSUB command is executed some values are placed on the stack and when a RETURN command is executed these values are removed). An "ERROR - OM" will occur if the programmer attempts to place too many items on the stack.

String Buffer - that area in memory which holds all character strings used in the BASIC language.

System - the internal network of Comm-Stor which guides the programmer's activities; or, the entire Comm-Stor unit.

Terminal Buffer - that area in memory used to temporarily store data sent from the device attached to the terminal port of the Comm-Stor unit.

User - that person, or group of persons, regulating and defining the operation and functions of Comm-Stor within a given working environment.

User Memory Space - All the internal memory available to the programmer for the allocation of buffer space.

Variable Space - that area of the User Memory Space consisting of the String Buffer, Free Space Buffer, and Array and Variable Buffer.

Whole Numbers - the set of natural numbers, their negatives, and zero. (...-3,-2,-1,0,1,2,3,...).

6. MEMORY ORGANIZATION

The internal memory of Comm-Stor is divided into two separate sections. One section is maintained and controlled solely by Comm-Stor itself. The other section is the User Memory Space and is maintained and controlled by the programmer. The programmer is able to specify, at configuration time, where the User Memory Space will begin and end. Therefore, the size of the User Memory Space can be managed by the programmer, providing a greater amount of flexibility when programming in the BASIC language.

User Memory Space is divided into 8 different areas called buffers. These buffers temporarily hold items of data which are operated on by BASIC commands. The size of these buffers and how they interact are discussed in Appendix B.

The organization of the User Memory Space is illustrated in Figure #1-3. For those Sykes BASIC commands which affect the User Memory Space, this type of diagram will be used as a descriptive aid.

Although these buffers are not always used by the programmer and, in fact, do not all need to exist, this diagram depicts how the User Memory Space is partitioned when all 8 buffers are present.

USER MEMORY

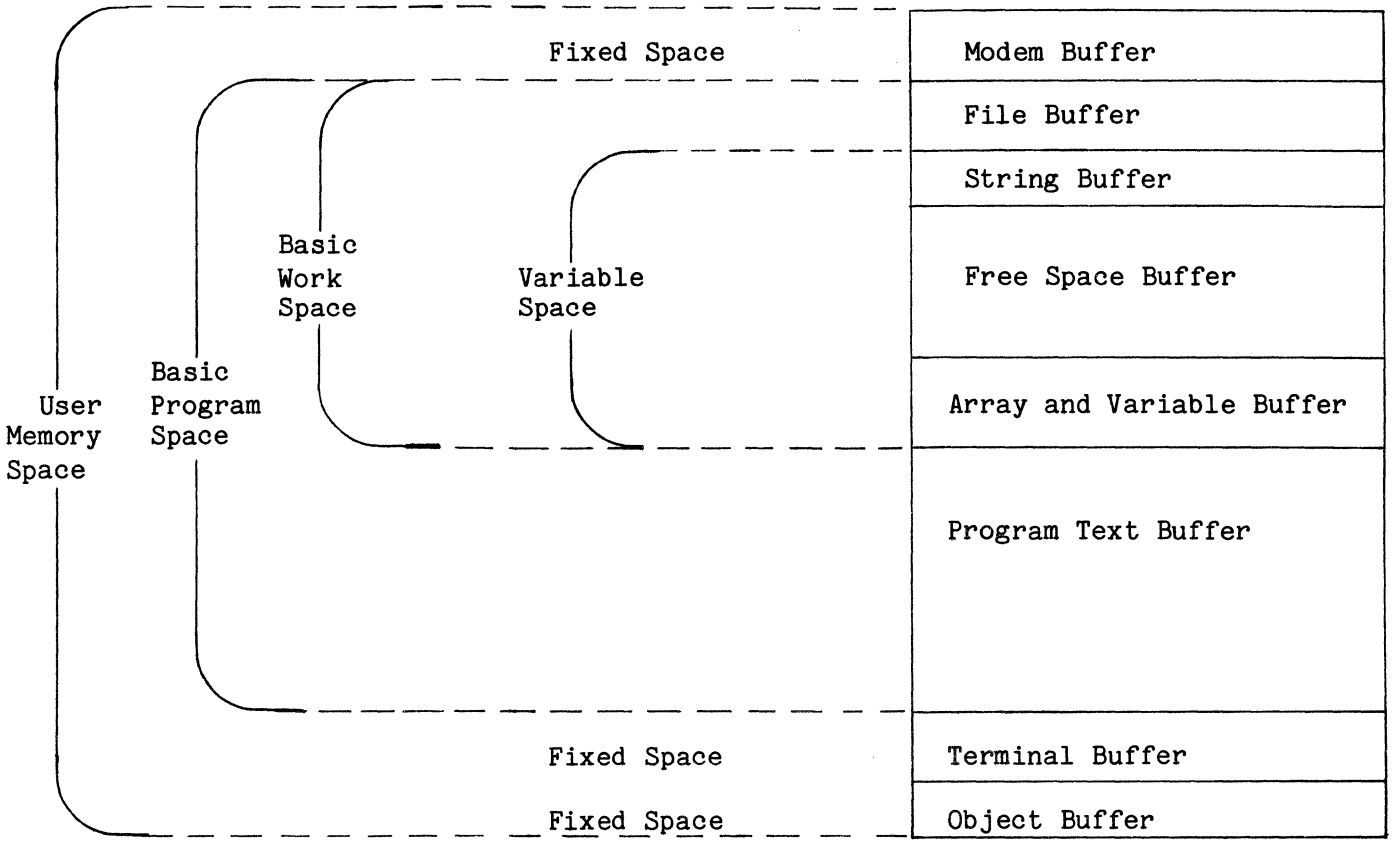


FIGURE #1-3

7. BASIC WITH Comm-Stor COMMANDS

A. General

The Comm-Stor system allows Comm-Stor commands to be issued from BASIC mode by the use of the BASIC command, "COM". Figure #1-4 illustrates the relationship between the Comm-Stor and the BASIC modes when the following commands are issued:

```
.E ABLE
   (TEXT)
[CNTRL/C]
.DD *
(DIRECTORY DISPLAY)
.BA
READY
>LOAD "TEST"
READY
>RUN
 .
 .
50 COM ".CN ABLE"
 .
 .
READY
>BYE
.DD *
```

There are many applications for executing Comm-Stor commands from BASIC mode; some of these are illustrated in the following examples.

B. Examples

Example I. This is a simple example which illustrates the ability to name disk files using a BASIC program.

```
.
.
.
230 INPUT "WHAT NAME DO YOU WISH FOR THIS
FILE";A$
```

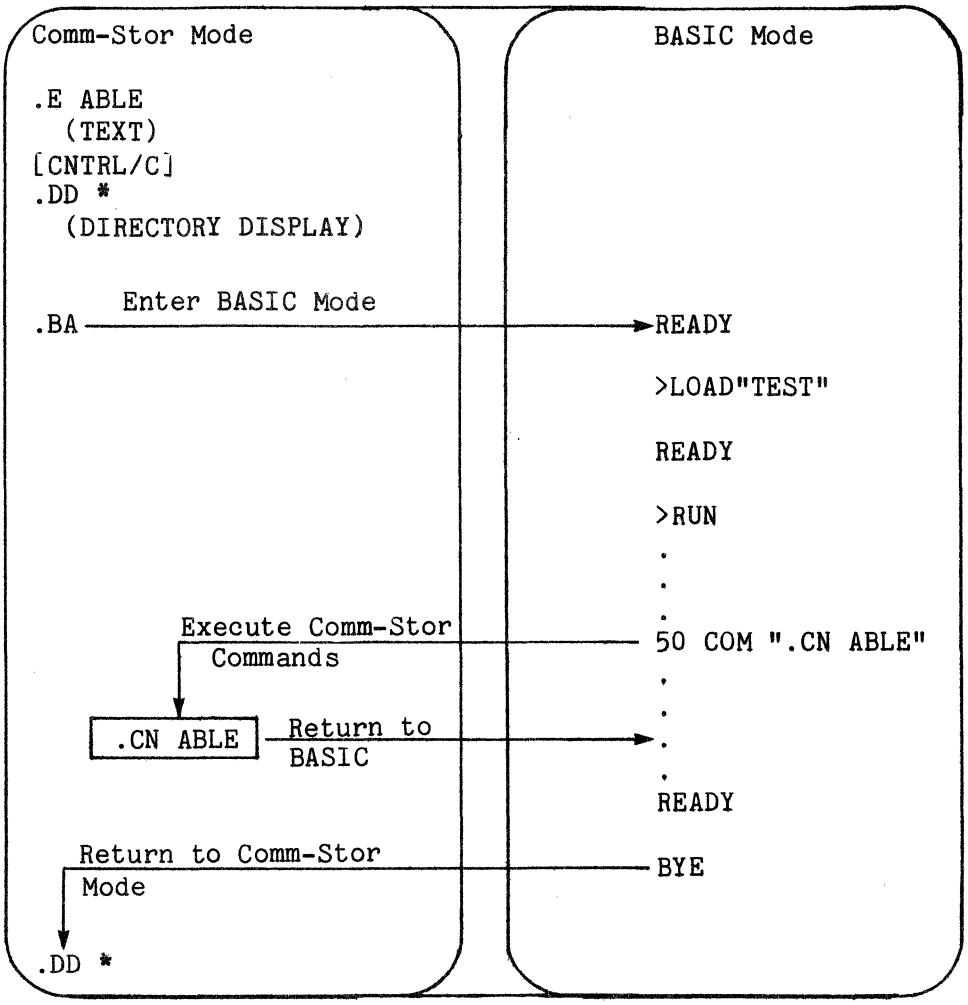


FIGURE #1-4

```
240 IF ERROR #25 GOTO 500
250 CREATE A$,M
260 ...
.
.
.
490 END
500 PRINT "A FILE ALREADY EXISTS BY THAT
NAME"
510 PRINT "HERE IS THE DIRECTORY OF DISK
FILES:"
520 COM ".DD *"
530 GOTO 230
```

Error number 25 (line 240) is associated with "ERROR - DE", which signifies a duplicate directory entry. If this error is detected by the program, the error message will not be displayed but control will be transferred to line number 500 which then issues a Comm-Stor command to display the entire directory. This allows the programmer to choose another file name which presently does not exist. Control is then returned to line number 230 to enter a different file name. This sequence of steps will be executed until a new file name is chosen that does not presently exist on Disk.

Example II. This example illustrates the use of the LOAD INITIAL command and the procedure for accessing the next file name that will be used to create a disk file using the .EA command.

```
1 CREATE "STATUS",M
2 OPEN#4 TO "STATUS"
3 COM".LI TEXT<001>ABC"
4 GOSUB 1000
5 PRINT "ENTER THE CONTENTS OF FILE";A$
6 PRINT "WHEN DONE ENTERING THE DATA, EN
TER A [CNTRL/C]"
7 COM".EA"
8 ...
.
.
.
850 GOTO4
.
.
.
990 END
1000 COM#4;".DS"
1005 REWIND#4
1010 FOR I=1 TO 5
1020 READ#4;A$
1030 NEXT I
1040 READ #4 USING "10#";A$
1050 RETURN
```

Upon execution of the GOSUB command (line 4), the disk file named "STATUS" will receive the output of the DISPLAY STATUS command from Comm-Stor mode. Line 7 is then read by Comm-Stor to obtain the next file name used with the next .EA command.

Example III. Using Dual Drive units (Special Application)

Suppose a company enters the current day's date as the EXTENSION of every disk file created. Suppose, too, that some of the 500 social security numbers entered in the month of December were incorrect. It is known that the social security numbers are in the form "XXX-XX-XXXX" and start in the thirty-sixth position of the first record in each file. It is also known that each file name contains 6 characters and that each extension is in the form of "MM--DD-YY". Instead of editing each one of the files, which could easily take a whole day for 500 files, a BASIC program like the one below could be executed to accomplish this task.

```
100 REM CREATE A DUMMY FILE FOR HOLDING
FILE NAMES
110 CREATE (2) "ERROR",M
120 OPEN #4 TO (2) "ERROR"
130 REM RESERVE FILE BUFFER SPACE FOR EACH
FILE
140 OPEN #5 TO +
150 RELEASE#5
160 REM PUT ALL APPLICABLE FILE NAMES INTO
THE DUMMY FILE
170 COM#4;".DD *+12-??-78"
180 REM GET BACK TO THE BEGINNING OF THE
DUMMY FILE
190 REWIND#4
200 REM DETECT END OF FILE NAMES
210 IF END #4 GO TO 460
220 REM SKIP OVER BEGINNING OF DIRECTORY
DISPLAY
230 FOR I=1 TO 3
```

```
240 READ#4;A$
250 NEXT
260 FOR I=1 TO 500
270 REM OBTAIN A LINE OF THE DIRECTORY
DISPLAY
280 READ#4;A$
290 REM GET THE FILE NAME
300 A$=LEFT$(A$,6)
310 REM UPDATE THE FILE
320 ASSIGN #5 TO A$
330 READ #5,1 USING"P(36)11#";S$
340 PRINT "THE SOCIAL SECURITY NUMBER FOR
FILE";A$
350 PRINT "IS";S$
360 INPUT "DO YOU WANT TO CHANGE IT";Z$
370 IF LEFT$(Z$,1)<>"Y"GOTO 430
380 J=J+1
390 REWIND #5
400 PRINT "WHAT IS THE SOCIAL SECURITY
NUMBER FOR FILE";A$;"?"
410 INPUT S$
420 PRINT#5,1 USING "P(36)11#";S$
430 CLOSE#5
440 NEXT
450 REM LAST FILE COMPLETE
460 PRINT "PROGRAM COMPLETE"
470 PRINT J;"FILES WERE UPDATED"
480 KILL (2) "ERROR"
```

Upon execution of this program, a directory of all files in the upper drive created in the month of December will be stored in the disk file named "ERROR" on the lower drive. Each record of this file is read to obtain a file name. The disk file corresponding to this file name is then read and the social security number stored in it will be displayed. If the programmer wishes to correct the social security number that file may be updated and the next file name is obtained. When there are no more file names left, control is transferred to line number 460 which displays the total number of updated files, cancels the disk file "ERROR", and automatically terminates the program.

8. PROGRAM LIMITS AND MEMORY OVERHEAD

Ranges

Integers - 8388608 to+8388608 exclusive

Single Precision - 1.7014118E+38 to +1.7014118E+38 inclusive

Double Precision -1.7014118346E+38 to +1.7014118346E+38 inclusive

String Range: Up to 255 characters

Line Numbers Allowed: 0 to 63999 inclusive

Program Line Length: Up to 255 characters

Memory Overhead

Program lines require 5 bytes minimum, as follows:

Line Number - 2 bytes
Line Pointer - 2 bytes
Carriage Return - 1 byte

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

Dynamic (RUN-time) Memory Allocation

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

Single precision variables: 7 bytes each
(5 for value, 2 for variable name)

Extended precision variables: 10 bytes each (8 for value, 2 for variable name)

String variables: 5 bytes minimum
(2 for variable name, 3 for variable pointers, 0 for a null string otherwise 1 for each character)

Array variables: 10 bytes minimum
(2 for variable name, 2 for size, 1 for number of dimensions, 2 for each dimension, and 3,5 or 8 [depending on array type] for each element in the array).

Each active FOR-NEXT loop requires 14-24 bytes depending on the type of control variable used. Each active (non-returned) GOSUB requires 6 bytes.

CHAPTER 2

CONSTANTS AND VARIABLES

CONTENTS PAGE

1. ASCII CHARACTER SET

Sykes BASIC uses the ASCII character set found in Appendix C of this manual. BASIC program lines* are composed of individual ASCII characters which fall under four categories: alphabetic, numeric, control and special characters.

*A BASIC program line is defined as one or more BASIC commands, where multiple commands are separated by colons.

A. Alphabetic Characters

The alphabetic characters in Sykes BASIC are the letters of the English alphabet, A through Z and a through z, inclusively, and a space.

B. Numeric Characters

The digits 0 through 9, inclusively, are numeric characters.

C. Control Characters

Control characters are nonprintable characters which perform a specific function when issued. Most of these control characters are issued by depressing the "CTRL" (for Control) key and the corresponding letter simultaneously, and are denoted in this manual by [CNTRL/letter]. The following is a list of the most commonly used control characters and their functions. The hexadecimal equivalent is provided for you in case the characters listed below do not appear on your terminal.

Hexadecimal Equivalent	Control Character	Function
03	[CNTRL/C]	Denotes the end of a Comm-Stor disk file.
09	[CNTRL/I]	Tab to next character (see Chapter 7, EDIT).
0E	[CNTRL/N]	Used for rejecting characters in Comm-Stor mode.
11	[CNTRL/Q]	Resume Display
13	[CNTRL/S]	Hold Display
14	[CNTRL/T]	Reset: returns system to idle state.

18	[CNTRL/X]	Cancels line being entered.
19	[CNTRL/Y]	Search for character (see Chapter 7, EDIT).
0A	[Line Feed] or [CNTRL/J]	Moves display up one line.
0D	[RETURN] or [CNTRL/M]	Denotes the end of a line.
1B	[ESC]	Obtain next character (see Chapter 7, EDIT).
7F	[DEL] or [RUB]	Deletes last character entered.

D. Special Characters

All ASCII characters which are not alphabetic, numeric or control characters are called SPECIAL CHARACTERS. The following is a list of special characters which have syntactical meaning in Sykes BASIC.

Character	Name
!	exclamation point
"	double quotation mark
#	number sign
$	dollar sign
%	percent sign
&	ampersand
(	left parenthesis
)	right parenthesis
*	asterisk or multiplication symbol
+	plus sign or addition symbol
,	comma
-	minus sign or subtraction symbol
.	decimal point
/	slash or division symbol
:	colon
;	semi-colon
<	"less than" symbol
=	"equal to" or "assignment" symbol
>	"greater than" or "BASIC prompt" symbol
<>	"not equal to" symbol
?	question mark
^ or ↑	exponentiation symbol
@	"at"sign or "NEW" symbol
⍽ or [SPACE]	

Spaces (or blanks) may be used freely throughout a BASIC program to improve readability. They have no syntactical meaning unless they are within character strings.

2. NUMERIC DATA

A. Constants

Numeric data (or NUMBERS) are composed of numeric characters which represent numeric values. Numeric values are either constants or variables. NUMBERS are often called numeric constants because they retain a constant value throughout a program. Numeric constants may be positive or negative; negative numbers must be preceded by a minus (-) sign. If a number is not preceded by a minus sign, its value is assumed to be positive.

Numeric constants may be entered and displayed in three formats: integer, fixed point and floating point.

Integer Format

Numbers expressed in integer format are written as a number of digits followed by a percent sign. The precision (maximum number of digits a number may contain) of an integer number, N, is seven digits, provided it is in the range -8388608<N <8388608.

Integer numbers do not contain a decimal point or fractional value. They are stored internally using three bytes (see Appendix E) and consequently, are processed much faster than fixed point or float-

ing point numbers. The numbers listed below are all examples of integer values:

5%
-255%
304678%

Fixed Point Format

Numbers expressed in fixed format may be written with a decimal point and fractional value. If a decimal point and fractional value are not specified, a decimal point is assumed to be to the immediate right of the right-most digit. A fixed point number, N, may be any number in the range $10^{-37}<N<10^{37}$. However, the accuracy of the number depends on the precision specified. There are two different precisions available for fixed point format: SINGLE PRECISION and EXTENDED PRECISION.

1. Single Precision

Single precision numbers have a precision of 8 digits. Although a single precision number, N, must be in the range $-1.7014118E+38 \leq N < 1.7014118E+38$, the maximum number of digits displayed is eight. Any single precision value which contains more than eight significant digits will be displayed using the floating point format (described below).

All numbers in Sykes BASIC are assumed to be single precision unless specified otherwise. That is, any number entered without being followed by a percent sign or number sign (see extended precision, below) will be assumed to be a single precision value. If desired, however, an exclamation point may follow a number to clarify it as a single precision number. The following are examples of single precision values.

5 or 5!
-255 or -255!
6.78 or 6.78!
543.56891 or 543.56891!

2. Extended Precision

Extended precision numbers are used for those values that are to be accurate to more than eight digits. The range of extended precision numbers is -1.7014118346E+38 to 1.7014118346E+38, inclusive. The precision of extended precision numbers is fourteen digits. When an extended precision value contains more than fourteen significant digits it will be displayed using floating point format (described below).

An extended precision number is any number followed by a number sign. The following are examples of extended precision values.

-5#
6.78#
543.56891#
98.53241567135#

Floating Point Format

The format of floating point numbers is:

(optional sign) mantissa E (optional sign) exponent (optional type)

The value of the mantissa is multiplied by 10 raised to the power of the exponent and is an integer, single precision, or extended precision value, depending on how the number was defined during entry. The percent sign defines integers, an exclamation point or nothing defines single precision numbers, and a number sign defines extended precision numbers. The mantissa may be a positive or negative whole or real number. The exponent may also be a positive or negative number, but it must be a whole number comprised of one or two digits.

This format corresponds to standard scientific notation in which numbers are expressed as a power of 10. For example, 5.23E4% is an integer number which means "5.23 times 10 raised to the power of 4" and is equivalent to 52350. Additionally, 7E-2 is a single precision number which

means "7 times 10 raised to the power of -2" which equals 7/100, or in BASIC notation, 0.07.

Floating point format may be used for entering integer, single precision, and extended precision numbers. However, the value must be within the precision and range of the type of number specified. For example, 8E10% is an illegal integer number since it lies outside the range for integer numbers.

Floating point format will only be used for display purposes when the number of significant digits in the value of a number exceeds the precision of the type of the number.

The following table contains some examples to further illustrate the use of floating point format:

Value Entered	Value Displayed
1E+6%	1000000
1E+6!	1000000
1E+6#	1000000
-1.5745E3%	-1574
-1.5745E3	-1574.5
-1.5745E3#	-1574.5
6.93E-1%	0
6.93E-1!	.693
6.93E-1#	.693
36.9854321E7%	ERROR - OV
36.9854321E7!	3.6985432E+08
36.9854321E7#	369854321

B. Variables

Although the value of a numeric constant may never be changed, a numeric variable is a named data item whose value may be changed, and therefore, may contain different values for different operations. Numeric variables are specified by a name and type (single, extended or integer precision). The name of a variable consists of one or more alphanumeric (alphabetic and/or numeric) characters, the first of which must be alphabetic. Additionally, no reserved words (Appendix D) may be embedded in a variable name. For example, LETTER and STORE are invalid names because the former contains the reserved word LET, and the latter contains the words TO and OR. Although any number of characters may be used in a variable name, only the first two characters are interpreted as defining the name; the rest are ignored. For example; CAR, CAT, CALL and CAN all represent the same variable name CA.

Numeric variables may be used to represent single precision, integer and extended precision values. Single precision is the default value for numeric variables. A single precision variable may be expressed by just the name or by the name followed by its respective type, in this case an exclamation point. The symbol for the type however, is not regarded as part of the name. It only associates the type of variable to the name. For example, A, A1!, C3 and XX! are all single precision variable names. An integer variable must be followed by a percent sign. For example, A%, A1%, C3% and XX% are all integer variable names. Similarly, if a number sign follows the variable name, such as A#, A1#, C3# and XX#, the variable name will be extended precision. Note that A, A% and A# all represent different variables; although they all have the same name, A, they each have different types.

Variables are assigned values by using the LET command (Chapter 7). For example:

```
LET A = 1
```

will assign the single precision value of 1 to the single precision variable A. The precision of the specified value is always set equal to the precision (or type) of the variable. For example,

```
LET A% = 1.5
```

will take the single precision value of 1.5, convert it to the integer value, 1, and assign that value to the integer variable A%. Note that before execution of a BASIC program begins, all variables are assigned the value of zero.

An added convenience of Sykes BASIC is the SET command (Chapter 7). This command enables the programmer to establish certain variables as a specified type without requiring the respective type symbol to follow each use of the variable name. For example, if the variables below are used in a program,

A%
BG%
E%
C3!
N!
K!
L2#
MAN#
Z#
D$
GG$
FX$

each time the variable name is used, its respective type must follow. However,

SET%A-E;!C;!K-N;#L;#M;#Z;$D;$F-G

may be used. This eliminates the need for the programmer to specify the type of variable in each application. After the SET command is executed, Comm-Stor will automatically keep track of those types of variables whose names begin with a specified letter. Below is a list of the names of those variables.

First Letter of Variable Name	Type
A	integer
B	integer
E	integer
C	single precision
N	single precision
K	single precision
L	extended precision
M	extended precision
Z	extended precision
D	character string
G	character string
F	character string

Furthermore, if the variable BC$ is to be used, specifying BC$ will distinguish this variable as a character string (Chapter 2) and not an integer.

C. Functions

Sykes BASIC includes system functions which perform a variety of commonly used mathematical operations. All arithmetic functions are performed as single precision values. Each is then evaluated for a single value, called its "argument", which produces a single result.

A description of the arithmetic functions is provided below. An asterisk denotes a function whose result is an integer value. All other functions result in a single precision value.

Function Name	Description
ABS(x)	Absolute value of x
ATN(x)	Arctangent (in radians) of x
COS(x)	Cosine (in radians) of
DEC(x$)	Decimal value of hexadecimal value contained in X$
EXP(x)	Natural exponent of x
*FRE(x)	Number of free bytes in memory
*INT(x)	Converts x to an integer number
LOG(x)	Natural logarithm of x
*PEEK(x)	Contents (in decimal) of address x
*POS(ref)	Current position of pointer in the file or device associated with the specified reference number
RND(x)	Random number between 0 and 1
*SENSE(x)	Sensing of condition x (-1 if true, 0 if false)
*SGN(x)	Sign of x (-1 if negative, 0 if zero, 1 if positive)
SIN(x)	Sine (in radians) of x
SQR(x)	Square root of x
TAN(x)	Tangent (in radians) of x

D. Expressions

An arithmetic expression may consist of a single element (a numeric constant; variable, or function), or it may be a combination of these separated by arithmetic operators.

Arithmetic Operators

An arithmetic operator separates two values and indicates the specific arithmetic operation to be performed. The five arithmetic operators are listed below:

Arithmetic Operators:

Operator	Example	Meaning
^(or↑)	X^Y	Raise X to the power of Y
*	X*Y	Multiply X by Y
/	X/Y	Divide X by Y
+	X+Y	Add Y to X
-	X-Y	Subtract Y from X

Negative Operator

The negative operator is a special arithmetic operation. While most arithmetic operators perform a task on two values, the negative operator may perform a task on a single value. The minus (-) sign is considered to be a negative operator when it does not separate two values. The result of the negative operator executed on a value is a value opposite in sign of the original value. For example, -(1) would result in a negative 1 and -(-1) would result in a positive 1.

Arithmetic Hierarchy

When more than one arithmetic operation is specified in a single expression, that expression is evaluated according to the hierarchy (or precedence) of the operators involved. Operations with a higher precedence, or priority level, are performed before those with a lower priority level. Operations which exist on the same level are performed as they are encountered from left to right. The hierarchy of the operators, from highest to lowest, is:

1. exponentiation
2. negative operator
3. multiplication and division
4. addition and subtraction

For example, the expression:

A+B*C^D/E

is evaluated as follows:

1. C is raised to the power of D;
2. B is multiplied by the result of the first operation;
3. the result of the second operation is divided by E;
4. the result of the third operation is added to A;

Parentheses are used to change the order of operations. Any operations enclosed in parentheses are performed before operations which are not. For example, if the product of B and C is to be raised to the power of D divided by E, the appropriate expression would be:

A+(B*C)^(D/E)

It would be evaluated as follows:

1. B is multiplied by C;
2. D is divided by E;
3. the result of the first operation (B*C) is raised to the power of the result of the second operation (D/E);
4. the result of the third operation is added to A.

Parentheses may be used, even though they are not required, to make the expression more readable and understandable.

E. Logical Operators

Logical operations are defined using logical operators. Logical operations are performed on arithmetic values in the IF...GOTO, IF...THEN and IF...THEN...ELSE commands (see Chapter 7). Logical operators may be performed on true (-1) values, false (0) values, and other arithmetic data. Below is a list of the logical operators and their meanings:

Operator	Example	Meaning
NOT	NOT A	The logical negative of A. NOT-1 equals 0 NOT 0 equals -1
AND	A AND B	The logical product of A and B. Equals -1 only if A and B equal -1.
OR	A OR B	The logical sum of A and B. Equals 0 only if A and B equal 0.

When other values, besides -1 and 0, are used with the logical operators, these values are always treated as integer values. Logical operations performed on integer values require the use of Binary Arithmetic. (If a value is negative, it is stored internally in the binary two's complement format.)

3. CHARACTER DATA

Character data is data without a numeric value. Sykes BASIC also processes, or manipulates, character data, or strings. A string is a sequence of characters which may be alphabetic, numeric or special characters (except double quotation marks). Like arithmetic data, character strings may be in the form of constants or variables.

A. Constants

Character constants are character strings enclosed in a pair of double quotation marks. The following are examples of valid character constants:

```
"ABC"
"1234"
"THE TOTAL OF ITEMS IS"
"!#$%&:;S"
```

The length of a character constant is the number of characters it contains including spaces. The length of a character string cannot exceed 255 or an error will result.

B. Variables

A character variable is a named item of character string data whose value may be changed. Character variables are named by any legal variable name (page 2-4, Numeric Variables) followed by a dollar sign. For example,

```
A$
A1$
C3$
XX$
```

are all character variable names. Before a BASIC program is executed, the lengths of all character variables are set equal to zero (i.e. they contain nothing).

Character constants may be assigned to character variables by using the LET statement. For example,

```
LET A$ = "ABC"
```

will assign the characters ABC to the variable A$. As with character constants, the length of the contents of a character variable may not exceed 255.

C. String Functions

Besides arithmetic functions (page 2-5), Sykes BASIC also contains system functions which operate on character strings. Each string function produces a single result. The string functions are listed and described below.

Function Name	Meaning
ASC(X$)	Decimal value of first character in X$.
CHR$(x)	Character which is the ASCII equivalent to decimal value x.
HEX$(x)	Converts x to a hexadecimal character string.
LEFT$(X$,x)	Leftmost x characters in X$
LEN(X$)	Length of X$

MID$(X$,x)	Characters of X$, starting at the xth character.
MID$(X$,x,y)	Characters of X$, starting at the xth character, for y characters.
RIGHT$(X$,x)	Rightmost x characters in X$
SRCH(X$,Y$)	Returns the first position in X$ where Y$ is found.
SRCH(X$,Y$,x)	Same as above but starts searching in the xth position.
STR$(x)	Character string of decimal value x
VAL(X$)	Decimal value of X$

D. Expressions

A character expression may consist of a single element (character constant, variable or function) or a combination of these separated by the string operator.

String Operator

The plus sign (+) is the string operator in Sykes BASIC. It separates two string values and specifies CATENATION. Catenation refers to items linked together. The result of the catenation of two or more strings is one character string which contains all of the specified characters in their respective order. For example,

```
LET A$="HERE"+" IS"+" THE"+" BOOK"
```

will assign the character string "HERE IS THE BOOK" to the variable A$.

Additionally,

```
LET A$ = "TEST"+ STR$(1)
```

will assign the character string "TEST 1" to A$.

4. RELATIONAL EXPRESSIONS

A RELATIONAL EXPRESSION compares the values of two arithmetic expressions or two character expressions. The specified expressions are evaluated and their results are then compared according to the relational operator specified. The relational expression is either true or false. Relational expressions are only used in BASIC programs within IF...GOTO, IF...THEN or IF...THEN...ELSE statements (Chapter 7).

A. Operators

The relational operators are special characters. They are listed below:

Operator	Meaning
=	equal to
<>	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

B. Numeric Expressions

When two numeric constants, variables, expressions, or a combination of these are used in a relational expression, the two values are compared. The following is a list of examples to further illustrate this idea.

Relational Expression	Result
1%=1	TRUE
1%=1#	TRUE
1=1.0	TRUE
1.5=1.55	FALSE
1.5<1.55	TRUE
1.5<=1.55	TRUE
1.5>1.55	FALSE
1.5>=1.55	FALSE
1.5<>1.55	TRUE
(22%/7%)=(22#/7#)	FALSE

C. Character Expressions

When applied to strings, relational operators indicate an alphabetic sequence. This sequence can be determined by referring to the ASCII Code Chart in Appendix C. A list of examples follows:

Relational Expression	Result	Meaning
"A"="A"	TRUE	"A" is equal to "A"
"A">""	TRUE	"A" is greater than a NULL string
"A">="A "	FALSE	"A" is less than "A "
"A"<"B"	TRUE	"A" is less than "B"
"A">="AA"	FALSE	"A" is less than "AA"

5. ARRAYS

In addition to the variables described previously, Sykes BASIC allows the use of ARRAYS, or subscripted variables. Subscripted variables provide the programmer with additional computing capabilities for dealing with lists, tables, matrices, or any set of related data items. In Sykes BASIC, arrays are allowed to contain up to 255 subscripts, or dimensions, provided each dimension does not exceed 32,767 items, or elements.

The name of an array may be any valid variable name followed by one or more numeric expressions separated by commas and enclosed in parentheses. The following are valid arrays:

```
A(1)  B3(10,25)  CAT(1,1,1,1,1,2) A$(5,6)
```

For example, a list might be described as A(I), where I varies from zero to five, such as:

```
A(0),A(1),A(2),A(3),A(4),A(5)
```

This allows the programmer to reference each of the six elements in the list as a one dimensional algebraic matrix, such as:

```
A(0)
A(1)
A(2)
A(3)
A(4)
A(5)
```

A two dimensional matrix B(I,J), can be defined similarly and is illustrated below:

```
B(0,0)  B(0,1)  B(0,2)  B(0,3)  ...  B(0,J)
B(1,0)  B(1,1)  B(1,2)  B(1,3)  ...  B(1,J)
B(2,0)  B(2,1)  B(2,2)  B(2,3)  ...  B(2,J)
B(3,0)  B(3,1)  B(3,2)  B(3,3)  ...  B(3,J)
  .       .       .       .     ...    .
  .       .       .       .     ...    .
  .       .       .       .     ...    .
B(I,0)  B(I,1)  B(I,2)  B(I,3)  ...  B(I,J)
```

Large arrays can also be defined in the same manner, but there may only exist one array for each array name used.

The dimensions of an array may be EXPLICITLY or IMPLICITLY defined. To explicitly define an array, the DIM command (Chapter 7) is used. To implicitly define an array, the number of dimensions in the array is set to the number of dimensions specified in its first operation. However, each dimension will only contain eleven (0 through 10) elements. For example, if array A was not defined in a DIM command and was used in its first application as:

```
A(3)=5
```

it would be a one dimensional array containing eleven elements, the fourth of which would have the value of 5. After an array has been dimensioned, either explicitly or implicitly, it cannot be dimensioned again, unless an ERASE command (Chapter 7) is issued for that array.

For arrays that contain character data, the maximum number of characters in each element is 255.

The elements of an array are referred to by the SUBSCRIPTS of an array. For example, the fourth element of the array A above was referenced by the subscript 3. All subscripts in the system normally start at 0. The number at which the subscripts start is called the BASE of the subscripts. This base may be changed by using the OPTION BASE command. For example, to change the base from 0 to 1, issue the following command:

```
10 OPTION BASE 1
20 DIM A(15)
```

When line 10 is executed, the base of all subscripts will be changed from 0 to 1. When line 20 is executed, array A will contain 15 elements: A(1),A(2),...,A(14), A(15). However, if line 10 is deleted, then array A will contain 16 elements: A(0),A(1),...A(14),A(15). The OPTION BASE command is described more fully in Chapter 7.

Virtual arrays may also be used in Sykes BASIC and are discussed in Chapter 5.

6. DISPLAYING CONSTANTS AND VARIABLES

Data is displayed on the terminal device by using a PRINT command. The format of the data may differ depending on the variable list specified in the PRINT command. For example:

```
PRINT A$;B$;C$
```

will display the contents of A$, B$, and C$ in order without any spaces separating the different character strings. However,

```
PRINT A$,B$,C$
```

will display the contents the A$, space to the next tabular position, display the contents of B$, space to the next tabular position, and display the contents of C$. These tabular positions are determined at configuration time. Let us assume the column width is 14 characters and the line width (also specified at configuration time) of the terminal is 80 characters. Therefore, each line on the terminal will be separated into column fields which can be illustrated as:

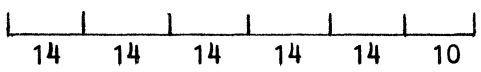


FIGURE #2-1

The tabular positions point to the first character position in each column field. Therefore, the tabular positions in this example are 15,29,43,57, and 71, and can be illustrated as:

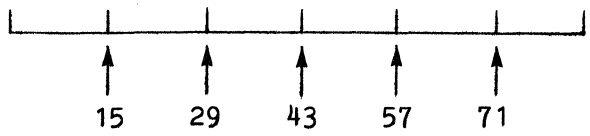


FIGURE #2-2

Thus, if:

```
PRINT "A","B","C";"D"
```

is executed, the display will be:

FIGURE #2-3

7. DERIVED FUNCTIONS

The following is a list of functions which can be derived from Sykes BASIC arithmetic functions.

Functions	Function Expressed in Sykes BASIC
Cosecant	1/SIN(X)
Cotangent	1/TAN(X)
Hyperbolic Cosecant	2/(EXP(X)-EXP(-X))

Hyperbolic Cosine	(EXP(X)/EXP(-X))/2
Hyperbolic Cotangent	EXP(X)/(EXP(X)-EXP(-X))*2
Hyperbolic Secant	2/(EXP(X)+EXP(-X))
Hyperbolic Sine	(EXP(X)-EXP(-X))/2
Hyperbolic Tangent	-EXP(-X)/(EXP(X)+EXP(-X))*2+1
Inverse Cosecant	ATN(1/SQR(X*X-1))+(SGN(X)-1)*1.5708
Inverse Cosine	-ATN(X/SQR(-X*X+1))+1.5708
Inverse Cotangent	-ATN(X)+1.5708
Inverse Hyperbolic Cosecant	LOG((SGN(X)*SGN(X*X+1)+1)/X)
Inverse Hyperbolic Cosine	LOG(X+SQR(X*X-1))
Inverse Hyperbolic Cotangent	LOG((X+1)/(X-1))/2
Inverse Hyperbolic Secant	LOG((SQR(-X*X+1)+1/X)
Inverse Hyperbolic Sine	LOG(X+SQR(X*X+1))
Inverse Hyperbolic Tangent	LOG((1+X)/(1-X))/2
Inverse Sine	ATN(X/SQR(-X*X+1))
Secant	1/COS(X)

This page intentionally blank.

CHAPTER 3

GETTING STARTED

CONTENTS PAGE

GETTING STARTED

The purpose of this chapter is to acquaint the programmer with Comm-Stor by actually using the system. It will include a discussion of some common BASIC commands, those commands particular to Sykes BASIC, and several examples to help the programmer become familiar with Comm-Stor.

1. COMMON BASIC COMMANDS

There are some BASIC commands within the computing industry which are prevalent in most BASIC language packages and Comm-Stor contains such a package. We are assuming in this manual that the programmer is familiar with these commonly used commands. If you wish to refresh your memory on the subject, refer to the book recommended on page 1-1.

The common BASIC commands are listed below. Please refer to Chapter 7 for their respective descriptions.

BYE	LET
CLEAR	LIST
CONT	LOAD
DATA	NEW
DEF	NEXT
DIM	PRINT
END	READ
FOR	REM
GOSUB	RESTORE
GOTO	RETURN
IF...GOTO	RUN
IF...THEN	SAVE
INPUT	STOP

2. SYKES BASIC COMMANDS

Besides the commonly used commands, Sykes BASIC provides some additional commands. These commands are briefly described below and are further described in Chapter 7.

APPEND - permits the programmer to enter additional data into an existing disk file. See Chapter 5.

ASSIGN - relates a disk file with a BASIC program, providing the File Buffer (see Appendix B) has been allocated. See Chapter 5.

AUTO - used to automatically supply line numbers when manually entering BASIC program lines into the system.

CALL - overlays a BASIC program on the current BASIC program and reloads the caller on execution of a RETURN.

CLEARV - clears variable space from User Memory Space. See Appendix B.

CLOSE - tells Comm-Stor that processing a disk file is complete. See Chapter 5.

COM - enables the programmer to execute a Comm-Stor command in BASIC mode.

CREATE - creates a disk file. See Chapter 5.

DEL - enables the programmer to delete a range of BASIC lines from User Memory Space.

DIM# - used for virtual arrays. See Chapter 5.

EDIT - allows the programmer to alter BASIC lines without having to re-enter entire lines.

ERASE - allows the programmer to delete specific arrays from User Memory.

IF END# - allows the programmer to detect and control an end of file condition. See Chapter 5.

IF ERROR# - allows the programmer to detect and control any error which might occur in the system. The error number is referenced in Appendix G.

IF...THEN...ELSE - allows the programmer to specify a certain action when a condition is true and another action if it is false.

IMAGE - specifies a format for inputting or outputting data using a PRINT USING, PRINT # USING, or READ # USING command. See Chapter 4.

KILL - deletes the specified disk file from a diskette.

LINK - allows the programmer to enter a new BASIC program into memory affecting only the Free Space and BASIC Text buffers.

MERGE - allows the programmer to add a BASIC program (saved in text format) to the current program in the User Memory Space. See Appendix B.

ON...GOSUB - transfers program control to different subroutines according to the specified value.

ON...GOTO - transfers BASIC program control to different line numbers according to the specified value.

OPEN - allocates additional space to the File Buffer (see Appendix B) for a disk file reference number and may also be used to change the configured size of the modem buffer. See Chapter 5.

OPTION BASE - changes the base of the subscripts currently being used.

POKE - changes a location in memory.

POSITION - allows for the random access of disk files. See Chapter 5.

PRESS - merges and strips a program of all REM commands and blanks.

PRINT USING - displays data according to a specified format. See Chapter 4.

PRINT # - outputs data to any device or disk file. See Chapter 5.

PRINT # USING - allows the programmer to output data according to a specified format to any device or disk file. See Chapter 4.

READ # - allows the inputting of data from any device or disk file.

READ # USING - allows data to be input according to a specified format from any device or disk file. See Chapter 4.

RECSIZE - changes the record size associated with a disk file reference number. See Chapter 5.

RELEASE - makes a portion of the File Buffer available for use. See Chapter 5.

RENUM - renumbers the current BASIC line numbers in User Memory Space.

REWIND - returns the accessing of a disk file to sequential mode, starting at the beginning of the file. See Chapter 5.

RTCS - enables or disables the Real-time Control System. See Chapter 6.

SET - declares certain variables as particular data types. See Chapter 2.

SETIME - specifies certain times which are to be used by the Real-time Control System or the System Clock. See Chapter 6.

SYSTEM - loads a Sykes-supplied object program into the Object Buffer.

TRACE - enables or disables the tracing of BASIC line numbers while the current program is executing.

TRUNCATE - deletes any unused space from a disk file.

UNLOCK - allows return to sequential accessing of a disk file. See Chapter 5.

USC - references a command located in the object buffer.

WAIT - temporarily stops the execution of the current program until a location in memory reaches the necessary operative value.

WATCH1 and WATCH2 - used for the hardware registers in the Real-time Control System. See Chapter 6.

WHEN - detects and controls certain interrupts specified in the Real-time Control System. See Chapter 6.

3. BASIC MODE

Comm-Stor offers two different modes to the programmer: Comm-Stor mode (refer to the Comm-Stor IV Reference Manual #9990B0258A) and BASIC mode. BASIC mode allows the programmer to operate in the BASIC programming language. BASIC mode consists of two modes: the DIRECT mode and the RUN mode.

A. The DIRECT Mode

When Comm-Stor is not executing a BASIC program, it is idle, or ready to receive a command. In this mode, the characters "READY" are displayed on the terminal. Additionally, the character ">" is displayed to remind the programmer that Comm-Stor is in the BASIC mode. Any BASIC command may now be executed directly (followed by a [RETURN] character telling the system to execute the command).

B. The RUN Mode

When Comm-Stor is in the RUN mode, it is executing a BASIC program. Issuing a [CNTRL/T] is the only way the programmer can interrupt the RUN mode and return Comm-Stor to the DIRECT mode.

4. Comm-Stor IV EXERCISES

We will now present some examples which we encourage the programmer to follow and execute.

Be sure that the Comm-Stor unit and the attached peripheral devices are powered on.

Exercise I. The following is a list of some BASIC commands and their explanations. Enter the indicated command statements, one at a time, and await the terminal display. Remember that a command will not be executed until a [RETURN] character is issued. Note: A question mark (?) entered without being enclosed in quotes represents the PRINT command.

Command Statement & Display	Explanation
.BA READY	instructs Comm-Stor to enter BASIC mode.
>PRINT2+3 5	determines the value of 2 plus 3 and displays it.
READY	
>?"HELLO" HELLO	displays the characters "HELLO"
READY	
>?TAB(20);"HELLO" HELLO	positions the printhead to the 20th position and displays the character string, HELLO.

Command Statement & Display	Explanation
READY	
>LET A=12	assigns the value of 12 to variable A.
READY	
>B=4	assigns the value of 4 to variable B.
READY	
>?A/B 3	divides the value of A by the value of B and displays the result.
READY	
>B=B+1	adds 1 to the value of B and assigns the result to B.
READY	
>?B 5	displays the value of B.
READY	
>A$="JOHN DOE"	assigns the characters "JOHN DOE" to the variable A$.
READY	
>?A$ JOHN DOE	displays the contents of A$.
READY	
>?LEFT$(A$,B) JOHN	displays the left-most B characters in A$ until the number of displayed characters equals the value of B.
READY	

Command Statement & Display	Explanation
>A$=LEFT$(A$,B)+"GREEN"	joins the left-most B characters in A$ with the characters "GREEN" and assigns the result to A$.
READY	
>?A$ JOHN GREEN	displays the contents of A$.
READY	
>B$=STR$(B)	converts the value of B to a character string and assigns it to B$.
READY	
>?B$ 5	displays the contents of B$.
READY	
>B=VAL(B$)+1	converts B$ to a numeric value, adds 1 to it, and assigns the result to B.
READY	
>?B 6	displays the value of B.
READY	
>10 REM THIS IS A REMARK	enters a REM command into the Program Text Buffer.
>20 ?"ENTER A NUMBER"	enters a PRINT command into the Program Text Buffer.
>30 INPUT A	enters an INPUT command into the Program Text Buffer.

Command Statement & Display	Explanation
>?INT(.23) 0	converts .23 into an integer value and displays it.
READY	
>40 Y=SQR(A)	enters a LET command into the Program Text Buffer.
>?ABS(-5)+1 6	determines the absolute value of -5, adds 1 to it, and displays the result.
READY	
>50 ?Y	enters a PRINT command into the Program Text Buffer.
>LIST	displays all BASIC lines in the Program Text Buffer.

```
10 REM THIS IS A REMARK
20 PRINT "ENTER A NUMBER"
30 INPUT A
40 Y=SQR(A)
50 PRINT Y
```

READY

>RUN	executes the BASIC lines which reside in the Program Text Buffer.

```
ENTER A NUMBER
?4.5
2.1213203
```

READY

>35 FOR I=1 TO A	enters a FOR command into the Program Text Buffer.
>40 Y=SQR(I)	enters a LET command into the Program Text Buffer.
>60 NEXT	enters a NEXT command into the Program Text Buffer.
>RUN	executes the BASIC lines which reside in the Program Text Buffer.

```
ENTER A NUMBER
?5
 1
 1.4142136
 1.7320508
 2
 2.236068
```

READY

>RUN 30	executes the BASIC lines which resides in the Program Text Buffer starting with line 30.

```
?2
 1
 1.4142136
```

READY

>

Now, insert a User diskette into the diskette drive (the upper drive of a dual drive unit).

>CREATE "TEST"	obtains the average file size on the diskette and assigns it to file "TEST".

READY

>OPEN #4 TO"TEST"	creates the File Buffer and assigns file "TEST" to file reference number 4.

READY

Command Statement & Display	Explanation
>PRINT #4 ;"THIS IS A TEST"	enters a character string into file "TEST".
READY	
>? #4;1,2,3,4	enters 4 numbers into file "TEST".
READY	
>CLOSE #4	deactivates file "TEST".
READY	
>ASSIGN #4 TO "TEST"	assigns file "TEST" to file reference number 4, using the same space in the File Buffer.
READY	
>READ #4;A$	obtains a character string from file "TEST".
READY	
>?A$ THIS IS A TEST	displays the character string.
READY	
>READ #4;A,B,C,D	obtains 4 numbers from file "TEST".
READY	
>?A,B,C,D 1 2 3 4	displays the four numbers.
READY	
>CLOSE	deactivates all files.
READY	
>?B 2	displays the value of B.
READY	

Command Statement & Display	Explanation
>CLEARV	deletes the contents of the Variable Space and returns these bytes to the Free Space Buffer.
READY	
>?B 0	displays the value of B.
READY	
>OPTION BASE 1	starts all subscripts at 1 instead of 0.
READY	
>DIM C$(2)	dimensions C$ to have 2 elements.
READY	
>C$(1)="HELLO"	assigns a character string to the first element of C$.
READY	
>C$(2)="GOOD BYE"	assigns a character string to the second element of C$.
READY	
>FOR I=1 TO 2:?C$(I):NEXT HELLO GOOD BYE	displays the contents of both elements of C$.
READY	
>ERASE C$	deletes array C$ from the Array and Variable Buffer and deletes the previous contents of the array from the String Buffer.

Command Statement & Display — Explanation

```
READY

>FOR I=1 TO 2?C$(I):NEXT
```
displays the contents of the first two elements in array C$.

```
READY

>LIST
```
displays all BASIC lines which reside in the Program Text Buffer.

```
10 REM THIS IS A REMARK
20 PRINT "ENTER A NUMBER"
30 INPUT A
35 FOR I=1 TO A
40 Y=SQR(I)
50 PRINT Y
60 NEXT

READY

>NEW
```
deletes everything in the BASIC Program Space and returns the bytes to the Free Space Buffer.

```
READY

>LIST
```
displays all BASIC lines which reside in the Program Text Buffer.

```
READY

>BYE
```
exits BASIC mode.

Exercise II. This exercise is designed to solve the common problem of determining compound interest. This problem is easily solved by the use of Sykes BASIC. Suppose we deposit a certain amount of money into a savings account. The savings account bears a certain interest rate which is compounded a certain number of times per year. How much money will we have at the end of a certain number of years?

Let A = the original amount of money, in dollars, that we deposited.
Let R = the annual interest rate, in %.
Let T = the number of times per year that the interest is compounded.
Let Y = the total number of years we will keep the money in the account.

First, enter:

```
.BA
```
to enter BASIC mode

```
READY

>
```

If we now enter:

```
>AUTO 1,1
```

the BASIC line numbers will be automatically supplied. If a mistake is made when entering the following BASIC lines, simply enter [CNTRL/X] and re-enter the same line.

Now, obtain the values for A, R, T, and Y. Enter:

```
1 INPUT "WHAT IS THE ORIGINAL AMOUNT";A
```

When this command is executed, the amount entered from the terminal will be assigned to the variable A. Enter:

```
2 INPUT "WHAT IS THE ANNUAL INTEREST RATE";R
```

When this command is executed, the amount entered from the terminal will be assigned to the variable R. Enter:

```
3 INPUT "HOW MANY TIMES PER YEAR IS THE INTEREST COMPOUNDED";T
```

When this command is executed, the value entered from the terminal will be assigned to the variable T. Enter:

```
4 INPUT "HOW MANY YEARS WILL YOU KEEP THE MONEY DEPOSITED";Y
```

When this command is executed, the number entered from the terminal will be assigned to the variable Y.

Now that the BASIC program is equipped to enter the values, it must calculate the answer. This requires three more variables.

Let N = the total number of interest periods.
Let R1 = the interest rate per period.
Let B = the final amount of money.

Enter:

```
5 N=Y*T
```

When this command is executed, the total number of years (Y) multiplied by the number of times per year (T) will be assigned to the variable N. Enter:

```
6 R1=R/100/T
```

When this command is executed, the annual interest rate (R) is divided by 100 resulting in a numerical percentage. That result is then divided by the number of times per year that the interest in compounded(T). That result will be assigned to the variable R1. (For example, if the annual interest rate is 12%, divide 12 by 100 to get the numerical percentage 0.12. Assume the interest is compounded once a month. We then divide 0.12 by 12 (12 months per year) and we have 0.01, or 1% interest each month. Next, calculate the final amount as follows:

final amount = original amount multiplied by 1% interest per period raised to the number of periods

We can do this by entering:

```
7 B=A*(1+R1)^N
```

The final amount will now be assigned to the variable B. Next, display the final amount. Enter:

```
8 PRINT "THE FINAL AMOUNT IS $";B
```

The program, however, is still not ready to be executed. What would happen if a zero or a negative number were entered as one or more of the values? If an answer appeared, it would be meaningless. Therefore, the program must have some logic in it which would check the values entered from the terminal. It would be best to have this check after the values have been entered, but before any calculations are attempted. Insert a line between lines 4 and 5. However, only whole numbers are allowed to be line numbers. Therefore, enter:

```
9[RETURN]

READY

>
```

the automatic supplying of line numbers will now be stopped. Enter:

```
>RENUM
```

and the line numbers will all be renumbered according to the default incrementation values. Now, enter:

```
>LIST

100 INPUT "WHAT IS THE ORIGINAL AMOUNT";A
110 INPUT "WHAT IS THE ANNUAL INTEREST
RATE";R
120 INPUT "HOW MANY TIMES PER YEAR IS THE
INTEREST COMPOUNDED";T
130 INPUT "HOW MANY YEARS WILL YOU KEEP
THE MONEY DEPOSITED";Y
140 N=Y*T
150 R1=R/100/T
160 B=A*(1+R1)^N
170 PRINT "THE FINAL AMOUNT IS $";B

READY

>
```

What was previously line number 4 is now line number 130 and line number 5 is now line number 140. Enter the command to check the entered values:

```
>135 IF A=0 OR R<=0 OR T<=0 OR Y<=0 GOTO
200
```

As this command is executed, if any values are equal to zero or are negative values, control will be transferred to line number 200, where a prompt to enter acceptable values will be displayed. However, since we do not yet have a line number 200 in the program, enter:

```
>200 PRINT "INCORRECT DATA.  PLEASE RE-
ENTER"
```

and:

```
>210 GOTO 100
```

so that control will be transferred back to line 100 and the programmer will then be able to enter valid values.

Finally, enter one more command into the program. To avoid displaying the above error message after the final values, as is presently the case, enter:

```
>180 END
```

and:

```
>LIST

100 INPUT "WHAT IS THE ORIGINAL AMOUNT";A
110 INPUT "WHAT IS THE ANNUAL INTEREST
RATE";R
120 INPUT "HOW MANY TIMES PER YEAR IS THE
INTEREST COMPOUNDED";T
130 INPUT "HOW MANY YEARS WILL YOU KEEP
THE MONEY DEPOSITED";Y
135 IF A<=0 OR R<=0 OR T<=0 OR Y<=0 GOTO
200
140 N=Y*T
150 R1=R/100/T
160 B=A*(1+R1)^N
170 PRINT "THE FINAL AMOUNT IS $";B
180 END
200 PRINT "INCORRECT DATA.  PLEASE RE-
ENTER"
210 GOTO 100

READY

>
```

It is evident that after the final amount is displayed, the program will be terminated. Execute the program by entering:

```
>RUN
WHAT IS THE ORIGINAL AMOUNT? 100
WHAT IS THE ANNUAL INTEREST RATE? 12
HOW MANY TIMES PER YEAR IS THE INTEREST
COMPOUNDED? 2
HOW MANY YEARS WILL YOU KEEP THE MONEY
DEPOSITED? 5
THE FINAL AMOUNT IS $179.08477

READY

>
```

RUN the program again and enter a zero or negative number to verify that the check routine worked.

Since the final amount contains .08477 cents, use the PRINT USING command to obtain a correct dollars and cents figure.

Enter:

```
>170 PRINT USING ";###.##"; "THE FINAL
AMOUNT IS $";B
```

This command establishes an image (see Chapter 4) that will be used to display the character string and the value of B. The semi-colon (;) in the image tells Comm-Stor to display the character string, as a regular PRINT command would do. The two number signs (#) following the decimal point (.) tell Comm-Stor to display the value of B using only 2 digits to the right of the decimal point; these 2 digits will be rounded. RUN the program again.

```
>RUN
WHAT IS THE ORIGINAL AMOUNT? 100
WHAT IS THE ANNUAL INTEREST RATE? 12
HOW MANY TIMES PER YEAR IS THE INTEREST
COMPOUNDED?  2
HOW MANY YEARS WILL YOU KEEP THE MONEY
DEPOSITED? 5
THE FINAL AMOUNT IS $179.08

READY
```

```
>
```

The program works correctly. RUN the program again, using a different value for the original amount.

```
>RUN
WHAT IS THE ORIGINAL AMOUNT? 1000
WHAT IS THE ANNUAL INTEREST RATE? 12
HOW MANY TIMES PER YEAR IS THE INTEREST
COMPOUNDED? 2
HOW MANY YEARS WILL YOU KEEP THE MONEY
DEPOSITED? 5
THE FINAL AMOUNT IS $******
READY
```

What happened?!? The value of B contains more than 3 digits to the left of the decimal point. Comm-Stor displayed asterisks to tell the programmer that the value of B is larger than what was provided for in the image. So, change the image by entering:

```
>170 PRINT USING";8#.##";"THE FINAL AMOUNT
IS $";B
```

The image will now be able to accommodate any value of B which is equal to or less than 99,999,999.99. RUN the program once again.

```
>RUN
WHAT IS THE ORIGINAL AMOUNT? 1000000
WHAT IS THE ANNUAL INTEREST RATE? 12
HOW MANY TIMES PER YEAR IS THE INTEREST
COMPOUNDED? 2
HOW MANY YEARS WILL YOU KEEP THE MONEY
DEPOSITED? 5
THE FINAL AMOUNT IS $1790847.70
READY

>
```

Now the amount is correct, but it is hard to read. Change the image so that commas can be inserted into the display of the value in order to make the amount more readable. Enter:

```
>170 PRINT USING ";##,###,###.##"; "THE
FINAL AMOUNT IS $";B
```

If the program is executed again, the amount will be easier to read.

```
>RUN
WHAT IS THE ORIGINAL AMOUNT? 1000000
WHAT IS THE ANNUAL INTEREST RATE? 12
HOW MANY TIMES PER YEAR IS THE INTEREST
COMPOUNDED? 2
HOW MANY YEARS WILL YOU KEEP THE MONEY
DEPOSITED? 5
THE FINAL AMOUNT IS $1,790,847.70
READY

>
```

RUN the program again using the original values.

```
>RUN
WHAT IS THE ORIGINAL AMOUNT? 100
WHAT IS THE ANNUAL INTEREST RATE? 12
HOW MANY TIMES PER YEAR IS THE INTEREST
COMPOUNDED? 2
HOW MANY YEARS WILL YOU KEEP THE MONEY
DEPOSITED? 5
THE FINAL AMOUNT IS $       179.08

READY

>
```

The dollar sign appears 8 character positions to the left of the value. A dollar sign may be put into the image so that it will "float" with the displayed value. Enter:

```
>170 PRINT USING ";$$,$$$,$$$.##";"THE
FINAL AMOUNT IS";B
```

This will make the dollar sign appear to the immediate left of the displayed value. RUN the program once again.

```
>RUN
WHAT IS THE ORIGINAL AMOUNT? 100
WHAT IS THE ANNUAL INTEREST RATE? 12
HOW MANY TIMES PER YEAR IS THE INTEREST
COMPOUNDED? 2
HOW MANY YEARS WILL YOU KEEP THE MONEY
DEPOSITED? 5
THE FINAL AMOUNT IS     $179.08

READY
```

>

Notice that although a comma follows the dollar sign in the image, it is not displayed. This is because Comm-Stor automatically keeps track of commas for the programmer. The PRINT USING command is discussed in greater detail in Chapter 4. Now suppose the programmer wishes to execute this program many times. Entering RUN every time can be tedious and time consuming. If the END command is changed to a GOTO command, this will solve the problem. Enter:

```
>180 GOTO 100
```

Instead of ending execution of the program when the final amount is displayed, the programmer will now be able to enter new values, but how will the program ever end? After the final amount is displayed, control will always be transferred to the INPUT command. Instead of making the programmer issue a [CNTRL/T] to interrupt execution, it would be much better if the program handled this automatically. Solve this by choosing an unlikely value for one of the variables and use it as a "flag" for stopping execution of the program. Use T and 99999. Enter:

```
131 IF T=99999 THEN END
```

When a value of 99999 is entered for the variable T, execution of the program will be terminated. Any other values for T will cause line 135 to be executed, and so on. Try executing the program until everything is clearly understood. After you are done using this program enter:

```
>NEW
```

to prepare for the next example.

Exercise III. This exercise illustrates how to SAVE and LOAD a BASIC program. Again, use automatically supplied line numbers. This time enter:

```
>AUTO 10
```

and enter the commands exactly as shown below:

```
10 PRINT"HELLO, MY NAME IS SAM."
20 INPUT S
30 S=S+1
40 PRINT S
50 [RETURN]

READY

>
```

LIST the program. (This is done to check that the lines were entered properly.)

You will now need to insert a User diskette into the drive (the upper drive of a dual drive unit) which contains either fixed length files or variable length files.

Call this program "TEST" and SAVE it by entering:

```
>SAVE "TEST"
```

followed by:

```
>COM".DD *"
```

which proves the program was entered on the diskette. Entering:

```
>NEW
```

(to delete the program from memory)

followed by:

```
>LIST
```

shows that the program is no longer in memory. Enter:

```
>LOAD "TEST"
```

to bring the program back into memory and enter:

```
>LIST
```

to prove that the program is once again in memory. Keep this program in memory for the next exercise.

Exercise IV. This exercise illustrates the procedures for modifying lines within a BASIC program. A line within a program can be changed simply by manually re-entering the entire line. However, it is often simpler and less time consuming to use the EDIT command. For example, to change "SAM" in line 10 (Exercise III) to "TOM", enter:

>EDIT 10

Line 10 is displayed and Comm-Stor is now waiting for a character to be entered. Enter:

S

and everything up to the first "S" in the line is displayed. So, we have:

10 PRINT "HELLO, MY NAME IS

Now enter:

TOM."

to replace the previous name with "TOM" and enter:

[RETURN]

to delete (discard) the remainder of the line and to complete the edit for that line. Enter:

>LIST 10

to make sure that line 10 is correct.

Now, to change the variable name "S" to "SZ", enter:

>EDIT 20-40

(because "S" appears in lines 20 through 40). Line 20 is displayed and Comm-Stor is once again waiting for a character to be entered. Entering:

S

will cause:

20 INPUT S

to be displayed. Simply entering:

Z[RETURN]

will cause a "Z" to follow the "S" and the edit for that line to be completed.

Line 30 is now displayed and entering:

S

once again, will cause:

30 S

to be displayed. Enter:

Z

to cause a "Z" to follow the "S". Entering:

[CNTRL/I]

will cause another portion of the line, up to the next occurrence of an "S" to be displayed. Consequently, the line is now:

30 SZ=S

Once again, enter:

Z

to cause a "Z" to follow the "S". To add 5 to "S", instead of 1, enter:

[CNTRL/Y]1

which causes another portion of the line, up to the next "1", to be displayed. To change this "1" to a "5", enter:

[DEL]

(DELETE or RUBOUT) to delete the "1" from the line. Enter:

5

to cause a "5" to be added to the line, followed by:

[CNTRL/I]

which causes the edit for that line to be complete. (Actually, another "1" is searched for in the line and because one does not exist, the edit for that line is completed.)

Line 40 is now displayed. Enter:

S

to obtain:

40 PRINT S

and enter:

Z[RETURN]

to add a "Z" to the line and to complete the edit for that line.

The program is now modified; enter:

>LIST

to verify that it is correct.

Now, simply entering:

>SAVE

to SAVE the program in its modified form, Comm-Stor remembers which file name was used with the LOAD command, and consequently,

TEST--SURE?

is displayed. This is to notify the programmer that a file already exists by the name "TEST". Entering:

Y

causes Comm-Stor to delete the existing file "TEST" and SAVE the current program in a file by that name.

Enter:

>LOAD "TEST"

followed by:

>LIST

to verify the program is SAVEd in its modified form.

To look at the contents of the file, enter:

>COM".D TEST"

The display is not very understandable. This is because the program was stored in packed format (some of the contents of the file are translated into binary representation for speed efficiency). To SAVE it in text format (ASCII format), enter:

>SAVE "TEST",T

(where the "T" stands for text). Comm-Stor will again ask "--SURE?" to be answered with a "Y" for YES. Once again, enter:

>COM".D TEST"

to see the contents of the file clearly.

Enter:

>RUN

HELLO, MY NAME IS TOM.
?6 [RETURN]
11

READY

>

Keep this program in memory for the next exercise.

Exercise V. Now, suppose that the programmer would like the time of day to be displayed each time the value of "SZ" is displayed. Since the Comm-Stor system uses a 24-hour clock, the display must be converted to the 12-hour format. First, tell Comm-Stor the current time. This is normally done in Direct mode when first powering up the unit, but is included as part of the program for this example.

Enter:

```
>1 INPUT"WHAT IS THE TIME OF DAY";T$
```

Now, enter the current time into the system; the time must be represented by a character string consisting of 6 numeric characters, HH MM SS. To assure that the time entered from the terminal is in the correct form, enter:

```
>2 IF LEN(T$)<>6 THEN PRINT "INCORRECT DATA. PLEASE RE-ENTER":GOTO 1
```

Now, enter the correct time into the system by:

```
>3 SETIME #0 TO T$
```

This tells Comm-Stor the current time. The system now automatically increments this time every second.

Enter:

```
>50 HOUR$=LEFT$ (TIME$(0),2)
>60 MIN$=MID$(TIME$(0),3,2)
>70 SEC$=RIGHT$ (TIME$(0),2)
```

Line 50 assigns the current hour to the variable HOUR$, line 60 assigns the minutes to MIN$, and line 70 assigns the seconds to SEC$. To convert this time to a 12-hour clock, check the value of HOUR$. It should be noted here that "120000" represents 12 noon and "000000" represents 12 midnight. Therefore, a value of HOUR$ greater than 12 indicates that it is after 12 noon. Enter:

```
>80 IF VAL(HOUR$)=0 THEN HOUR$="12":W$="A":GOTO 130
>90 IF VAL(HOUR$)<12 THEN W$="A":GOTO 130
>100 IF VAL(HOUR$)=>12 THEN W$="P":GOTO 120
>110 W$="P"
>120 HOUR$="0"+RIGHT$(STR$(VAL(HOUR$) - 12),1)
>130 PRINT "THE CURRENT TIME IS ";HOUR$;":";MIN$;":";SEC$;"";W$;".M."
```

Lines 80 through 120 convert the hours value to that which corresponds to a 12-hour clock. Line 130 displays the time according to this format.

The lines entered in this exercise were added to the program from Exercise IV. Execute the program:

```
>RUN
WHAT IS THE TIME OF DAY? 000000
HELLO, MY NAME IS TOM.
? 6
11
THE CURRENT TIME IS 12:00:03 A.M.

READY

>RUN
WHAT IS THE TIME OF DAY? 134559
HELLO, MY NAME IS TOM.
?2
7
THE CURRENT TIME IS 01:46:00 P.M.

READY

>RUN
WHAT IS THE TIME OF DAY? 090021
HELLO, MY NAME IS TOM.
?245
250
THE CURRENT TIME IS 09:00:23 A.M.

READY

>NEW

READY

>
```

Note: The clock used by Comm-Stor may be used by the functions TIME, TIME$, CTIME, and CTIME$ which are described in Chapter 8. Also, there are two software registers that may be used by the programmer. These registers are called TIMER 1 and TIMER 2; each can hold a specific time. The times are entered into the registers by using the SETIME command the same way it was used above, where SETIME #1... is used for TIMER 1 and SETIME #2... is used for TIMER 2. Additionally, the clock functions may be used on these registers, where the argument is 1 for TIMER 1 and 2 for TIMER 2. The clock may further be used in conjunction with the Real-Time Control System, described in Chapter 6.

5. ERRORS

In Appendix G, there are 47 possible errors listed. These errors are the only ones which can occur in Sykes BASIC. A special feature of Sykes BASIC is that of allowing the programmer to control BASIC program execution when an error occurs. This is done by using the IF ERROR # command, where any whole number (between 0 and 49, except 10, 15, and 29) follows the "#" character and corresponds to a particular error. For example:

```
IF ERROR #7 GOTO 500
```

will cause program control to be transferred to line 500 any time an "ERROR - FULL DSK" occurs. The error message will not be displayed on the terminal when this command is used. This command (entered into a program in anticipation of a possible error) must be executed prior to Comm-Stor's detection of the error.

Additionally, the line number and error number specified in this command may be numeric variables. This allows greater flexibility for programming purposes. For example, in executing:

```
10 T=500
20 FOR I=0 TO 49
30 IF I=10 OR I=15 OR I=29 GOTO 60
40 T=T+1
50 IF ERROR #I GOTO T
60 NEXT I
```

each error in the system is associated with a particular line number in the BASIC program enabling the programmer to know which error actually occurred.

If the programmer wishes to stop control of a particular error, the same IF ERROR # command should be executed again by specifying "END" in place of the line number. For example:

```
IF ERROR #7 GOTO END
```

This will return error detection control entirely back to the system for an "ERROR - FULL DSK".

This page intentionally blank.

CHAPTER 4

PRINT USING AND READ USING COMMANDS

CONTENTS PAGE

1. THE IMAGE

As described in Chapters 2 and 3, Comm-Stor automatically formats data in output and input operations. Sykes BASIC provides the programmer with a simple, yet efficient way of controlling the formats used. This method of control is called an IMAGE.

The image shows the exact format of data which is output or input. Sykes BASIC allows the programmer three ways to specify an image:

1.	String literal	"#.##"
2.	String variable	A$="#.##"
3.	IMAGE command	50 IMAGE #.##

An image may contain any of the following options:

#	- field position
n#	- repeating field position
S	- space fill-in (Print Using)
nS	- repeating space fill-in
S	- skip feature (Read Using)
nS	- repeat skip feature
N	- null
nN	- repeating null fill-in
()	- repeat feature
+,-	- sign control
$	- floating dollar sign control
n$	- repeating dollar sign
.	- decimal point control
B	- binary (transparent) format
P	- reposition function
T	- truncate field character
E	- exponential notation
I	- insertion character
D	- delimiter assignment
A	- delimiter allocation
X	- suppress echo at terminal

An image specified in any of the above formats may be used in different ways, depending on the BASIC command in which it appears. There are three different commands in which an image may be used:

1. PRINT USING
2. PRINT # USING
3. READ # USING

These commands and their use of images are described below.

2. PRINT USING

The PRINT USING command may be employed in situations where a specific output format is desired. The format of the PRINT USING command is described in Appendix A. Like the regular PRINT command, the variable list is a list of variables, values, or expressions separated by commas and/or semi-colons.

The image for a particular variable, or value, is called a field. Each field must contain at least one number sign (#) for its corresponding value. If more than one number sign is needed, they may be explicitly or implicitly specified.

Example:

Explicit Image	Equivalent Implicit Image
###	3#
####.###	4#.###
4#.###	4#.3#

When the field for a string is not longer than the length of the string, such as:

```
PRINT USING "10#";"TEST"
```

the string will be displayed and followed by space characters until the image for that field is used entirely. In this case, the output would be:

TESTbbbbbb

(where the character "b" signifies a space character) because the length of "TEST" equals 4 and 4+6=10.

If the field for a string is shorter than the length of the string, such as:

```
PRINT USING "3#";"CHAIR"
```

only part of the string, starting from the left, will be displayed. In this case, the output would be:

CHA

because the length of "CHA" equals 3.

The string will be fully displayed if the field for a string equals the length of the string.

The field for a whole number may be longer than the number of character positions in its value, such as:

```
PRINT USING "5#";3
```

After the number of character positions in the value is determined, any unused character positions in the field, to the left of the value, will be replaced by blanks. In this case, the output will be:

bbbb3

since 3 uses one character position, 4 positions in the field are unused.

If the field for a whole number is shorter than the number of characters in its value, such as:

```
PRINT USING "#";10
```

the entire number of positions in the field are replaced by asterisks (*). In this case, the output would be:

*

because 10 uses two character positions, but the field only contains one.

If the field for any numeric value equals the number of characters in its value, the entire number will be displayed.

When a numeric contains a decimal point, such as the number 5.48, all digits to the left and right of the decimal point will be aligned with the decimal point in the field, If there is no decimal point in the field, the real value will be rounded to a whole number and displayed right justified.

If there are more character positions in the field to the right of the decimal point then there are in the value, such as:

```
PRINT USING "##.4#";5.48
```

zeros will follow those characters filling the field. In this case, the output will be:

b5.4800

since there are four character positions to the right of the decimal point in the field, but the value only contains two.

Should there be less character positions to the right of the decimal point in the field than there are in the value, such as:

```
PRINT USING "##.#";5.48
```

only that number of character positions to the right of the decimal point in the field will be displayed. However, the right-most character displayed will be rounded. In this case, the output will be:

Ø5.5

since the field only allows for one character to the right of the decimal point to be displayed. If the above numeric value was 5.43, instead of 5.48, the display would be:

Ø5.4

Truncate

The digit farthest to the right of the decimal point in any numeric value does not have to be rounded. Specifying a "T" in the beginning of the field will TRUNCATE (drop off or ignore) the extra characters. Such as,

```
PRINT USING "T##.#";5.48
```

will display:

Ø5.4

When a numeric field only contains values to the right of the decimal point, such as .56,

```
PRINT USING "#.##";.56
```

will display:

0.56

and

```
PRINT USING ".##";.56
```

will display:

.56

Table #4-1 further illustrates the relationships between decimal values and images.

If a numeric value is negative, a character position to the left of the decimal point must be included in the field in order to display the entire value. For example,

```
PRINT USING "2#.#";-5.6
```

will display:

-5.6

but:

```
PRINT USING "#.#";-5.6
```

will display:

***

Additionally,

```
PRINT USING "2#.##";-.12
```

will display:

-0.12

and:

```
PRINT USING "#.##;-.12
```

will display:

-.12

Sign Control

A field may contain a plus sign (+). In which case,

```
PRINT USING "+#.#";5.6
```

will display:

+5.6

Image	Value	Output
###.##	.52	550.52
.##	.67	.67
.#	.67	.7
T.#	.67	.6
###	25	5525
###.##	25.5	525.50
##.##	350	*****
##	12.5	13
##.#	42.79	42.8
##.#	250.5	****

TABLE #4-1

and:

```
PRINT USING "+#.#";-5.6
```

will display:

-5.6

Additionally, a field may contain a negative sign (-), such as:

```
PRINT USING "-#.#";-5.6
```

will display:

-5.6

and:

```
PRINT USING "-#.#";5.6
```

will display:

Ø5.6

If the field is longer than the numeric value, with no sign indicated in the image, and the numeric value is negative, such as:

```
PRINT USING "5#.2#";-66.35
```

the display will be:

ØØ-66.35

However,

```
PRINT USING "+4#.2#;-66.35
```

will display:

-ØØ66.35

Some instances may require the sign of a value to follow the number, instead of preceding it. For example:

```
PRINT USING "#.#+";5.6
```

will display:

5.6+

and

```
PRINT USING "#.#+";-5.6
```

will display:

5.6-

Table #4-3 contains some examples to further illustrate the use of positive and negative signs in PRINT USING images.

Exponential Notation

To display a numeric value in floating point format (Chapter 2) or scientific notation, an "E" may be specified in the field. For example,

```
PRINT USING "E#.##";31.415
```

will display:

3.14E+01

In this format, the field should contain only enough character positions to be displayed to the left of the "E". As stated previously (page 4-3) any characters that are not printed will be rounded unless a "T" is specified in the field. Several examples are shown in Table #4-2.

		Image	Value	Output
TE##.#	OR	E##.#	31.465	31.5E+00
		ET##.#	31.465	31.4E+00
		E#	1234	1E+03
		E.#	1234	.1E+04
		E#.#	.0000079	7.9E-06

TABLE #4-2

		Image	Value	Output
		+###	125	+125
		+###	-125	-125
		-###.#	-113.5	-113.5
		-###.#	113.5	␣113.5
		+###	45	+␣45
		+###	-45	-␣45
		-###	45	␣␣45
		-###	-45	-␣45
		##-	-26	26-
		##-	26	26␣
		##+	26	26+
		##+	-26	26-
-T##.#	OR	T-##.#	12.48	␣12.4
-T##.#	OR	T-##.#	-12.48	-12.4
+T4#.#	OR	T+4#.#	12.48	+␣␣12.4
+T4#.#	OR	T+4#.#	-12.48	-␣␣12.4
		#.#-	.56	0.6␣
		#.#-	-.56	0.6-
		T#.#-	.56	0.5␣
		T#.#-	-.56	0.5-

TABLE #4-3

The Dollar Sign

In many instances, dollar signs ($) must be displayed with a numeric value.

One dollar sign may be placed into a field, such as:

```
PRINT USING "$###";2
```

The display will be:

$␣2

All the rules mentioned previously hold true for this option.

If the number of characters in the value are one greater than the specified number of positions in the field such as:

```
PRINT USING "$#";12
```

the display will be:

12

and:

```
PRINT USING "$.##";-.12
```

will display:

-.12

Note that in both cases the $ has been ignored. However, if the number of characters in the value is two or more greater than the specified number of positions in the field, such as:

```
PRINT USING "$#";-12
```

the display will consist of asterisks.

**

Floating (+) and (-) Signs

Plus (+) and minus (-) signs "float" with dollar signs. Any unused character positions to the left of the decimal point will be replaced by space characters such as:

```
PRINT USING "$$#";2
```

In which case, the display will be:

␣$2

```
PRINT USING "$$.##";-.12
```

will display:

-$.12

Table #4-4 lists some examples which illustrates the use of floating (+) and (-) signs and dollar signs.

Insertion Option

Commas may be placed within a field to make a numeric value easier to read, such as:

```
PRINT USING "#,###";1234
```

		Image	Value	Output
		$##	123	123
		$$#	123	123
		$#	-12	**
		$###	12	$Ø12
4$#	OR	$$$$#	12	ØØ$12
3$#.#	OR	$$$#.#	32.65	Ø$32.7
		-$$#	-5	-Ø$5
		-$$#	5	ØØ$5
		$$#.##+	5.367	Ø$5.37+
		$$#.##-	-5.367	Ø$5.37-
		T$$#.##-	5.367	Ø$5.36Ø
		T$$#.##-	-5.367	Ø$5.36-
		3$###.##	1.20	Ø$ØØ1.20

TABLE #4-4

In this case, the display will be:

1,234

A comma will not be displayed if it is to the left of the most significant digit. For example:

```
PRINT USING "#,###";123
```

will display:

ØØ123

Further examples are listed below:

Image	Value	Output
#,###,###	1234567	1,234,567
#,###,###	123456	ØØ123,456
#,###,###	123	ØØØØØØ123
$$,$$#.##	1234.56	$1,234.56
$$,$$#.##	1.15	ØØØØ$1.15

Other characters may also be inserted within a field for a numeric value. For example,

```
PRINT USING "I(THIS IS TEST)###";123
```

will display:

THIS IS TEST123

If the value above was -123, the display would have been:

****

A maximum of 255 characters is allowed for each "I" specified. Characters may not be inserted into a numeric field to the right of the decimal point.

For example:

Image	Value	Output
###I(-)###	1008125	1008-125
##I(PDB)##	1213	12PDB13

Multiple Field Images

One image may contain many fields. In this case, the fields must be separated by semicolons (;). These semicolons do not display anything; they are only used to permit Comm-Stor to distinguish one field from another. Each field is correspondingly matched with the values or variables in the variable list. If there are more fields specified in the image than there are values, the unused or extra fields will be ignored. If the image contains less fields than the number of values, the image will be repeated until all of the values have been displayed.

The variable list in the PRINT USING command may affect the display. If the variables are separated by commas, Comm-Stor will automatically space to the next tabular position (described in Chapter 2) before displaying the specified value. If the variables are separated by semicolons, no automatic spacing by Comm-Stor will take place; the display will be formatted exactly as specified in the image.

Some examples (using a column width of 14 characters) are listed below:

```
PRINT USING "10#;3#";"TEST";"CHAIR"
TESTØØØØØØCHA
```

```
PRINT USING "5#;#;2#";3,10
ßßßß3*

PRINT USING "##.4#;##.#";5.48,5.48
ß5.4800ßßßßßßßß5.5

PRINT USING "#";1,2
1ßßßßßßßßßßßß2

PRINT USING "#";1;2
12
```

Space Fill-In

The last example reveals the potential difficulties in distinguishing one value from another. An "S" specified in a field will display a space for clarification. For example:

```
PRINT USING "#;S#";1;2
```

will display:

```
1ß2
```

More than one "S" may be specified for each space desired within a field, such as:

```
PRINT USING "#;3S#";1;2
```

the display will be:

```
1ßßß2
```

Null

An "N" may be specified in the beginning of a field to display a NULL character. More than one may be specified, if desired. A null is something which contains absolutely nothing. Some devices, however, contain a character which represents a null. This technique can be very useful when communicating with other devices attached to the modem port (see PRINT # USING, below).

For example:

```
PRINT USING"#;10N#";1;2
```

will display on most terminals:

```
12
```

If the device in use represented a NULL by using an "at" sign (@), the display from the above PRINT USING command would be:

```
1@@@@@@@@@@2
```

Delimiter

After each PRINT USING command is executed, the printhead or cursor pauses immediately after the last displayed character. The programmer may then issue a [RETURN] after the values are displayed. This [RETURN] is called a DELIMITER. (The delimiter may be configured, but the default value is a [RETURN] character.) A delimiter is used to separate different data items.

To issue this delimiter at the end of the display, a "D" should be specified at the end of the last field in a line. For example:

```
PRINT USING "#;3#D";1;"ABC"
PRINT USING "$#D";2
```

will display:

```
1ABC
$2
```

followed by a [RETURN]. If the "D"s were not used, such as:

```
PRINT USING "#;3#";1;"ABC"
PRINT USING "$#";2
```

the display would be:

```
1ABC$2
```

and the printhead or cursor would pause immediately after the 2.

The delimiter may also be issued before the last value is displayed by specifying "D" at the end of a particular field. For example:

```
PRINT USING "#D;##;###D;##";1;2;3;4
```

will display:

```
1
ß2ßß3
ß4
```

and the printhead or cursor will pause immediately after the 4 since a "D" was not specified in that field. In some instances, the delimiter may temporarily be changed to a different character. Specifying "A" with an argument will change the delimiter for a particular field from the current value to the specified argument. The argument must be one of the decimal values listed in Appendix C.

For example:

```
PRINT USING "A(44)#D;##;A(43)3#D;##D";1;
2;3;4
```

will display:

```
1,ß2ßß3+ß4
```

followed by a [RETURN]. This delimiter is for one-time use only and will be erased from memory after the field is displayed. That is, the comma and the plus sign in the above display will once again be regular characters and not delimiters.

Repeating Field Position

We have shown that particular options in a field may be repeated by specifying the number of desired repetitions in the option statement, such as: "10#", "2S", "5N" and "6$". Additionally, entire fields may be repeated. This is accomplished by surrounding the field, or fields, with parentheses and specifying the number of desired repetitions in front of the parentheses. Table #4-5 lists several examples.

Image	Equivalent Image
2#;2#;2#;3#	3(2#;)3#
2#;3S#;2#;3S#;5#	2(2#;3S#;)5#
2#;4#;S#;4#;S#;4#;S#;#;	2#;3(4#;S#;)#;

TABLE #4-5

This completes the discussion of the PRINT USING command. The following further illustrate the use of this command:

```
PRINT USING "15#";"THIS IS AN EXAMPLE"
THIS IS AN EXAM

I$="11#;$2#;S7#"
PRINT USING I$;"THIS COSTS";25;"DOLLARS"
THIS COSTS $25 DOLLARS

300 IMAGE9#;A(46)S2(3#,)3#D
310 PRINT USING 300;"THERE ARE";200000000
THERE ARE 200,000,000.

PI=3.14159
PRINT USING "5#;S#.5#";"PI=";PI
PI = 3.14159

A$="PI, WHEN TRUNCATED, EQUALS"
PRINT USING "26#;ST#.4#";A$;PI
PI, WHEN TRUNCATED, EQUALS 3.1415

A$="WHEN ROUNDED, PI"
PRINT USING "16#;#;#.4#";A$;"=";PI
WHEN ROUNDED, PI=3.1416

200 IMAGE19#;SE#.5#
PRINT USING 200; "PI, IN E NOTATION =";PI
PI, IN E NOTATION = 3.14159E+00
```

3. PRINT # USING

As described on page 3-2, the PRINT # command is used to output data to a disk file or an external device. With that command, Comm-Stor automatically formats the output. With the PRINT # USING command, however, the programmer may specify the format used to output the data.

The PRINT # USING command operates similarly to the PRINT USING command, except that the output is sent to a disk file or external device.

Reposition Function

The Reposition (P) option can be specified in an image only when accessing disk files randomly (see Chapter 5).

For example:

```
PRINT #4,2 USING"P(12)3#";"ABC"
```

will cause the pointer to move to the 12th character position in the second record of the file associated with file reference number 4. The characters "ABC" will then be entered into the file and no other part of the file will be affected. This is extremely useful for file maintenance or updating purposes.

Binary Option

The binary option, "B", is a useful tool for communicating with external devices and for optimizing the use of disk space. For example, many modems receive and send data continuously, and special characters are often included in this data. Specifying a "B" in a field inhibits Comm-Stor's automatic conversion to ASCII code when outputting data; therefore, the output format of the data will be the internal representation of the specified values. The following is a list of the different types of values available in Sykes BASIC and the number of bytes used by each of them:

Value	Number of Bytes
Integer	3
Single Precision	5
Extended Precision	8
One Character	1

Please refer to Appendix E for the internal formats of these values. When a "B" alone is specified in a field, the number of bytes for the corresponding data type will be output. For example,

```
PRINT #4 USING "B";I%,I!,I#,I$
```

will cause the data to be output as 3 bytes for I%, 5 bytes for I!, 8 bytes for I#, and LEN(I$) bytes for I$.

Number signs (#) may also be specified after the "B". When this is done, different results can occur, depending on the data types being used. When using character strings, only the number of characters will be output as specified by the number of number signs in the image. If more number signs are specified than the length of the string, nulls will be output until the image specifications are fulfilled. For example,

```
PRINT #4 USING "B5#";A$
```

will output the left-most 5 bytes contained in A$. Comm-Stor will not verify the value of the bytes when they are output.

When using integer data, "B#" will output the low order byte (Appendix E), "B2#" will output the low and middle order bytes, and "B3#" will be equivalent to "B". If more than 3 number signs are specified for integer values, nulls will be output until the image specifications are fulfilled. For example:

```
PRINT #4 USING "B10#";I%
```

will output 3 bytes for the value of I% followed by 7 nulls. The "B" option for integer values is a useful tool for conserving disk space and for increasing the rate at which data is transferred. For example, if we know the value of I% is equal to or greater than zero and equal to or less than 255, we know that only one byte, the low order byte, is needed to hold this value. Therefore, if we execute:

```
PRINT #4 USING "B#";I%
```

we will not only be saving 2 bytes of disk space, but we will also be increasing the rate of data transfer by 2/3. This is because executing:

```
PRINT #4 USING "B";I%
```

would also output the high and middle order bytes, which are equal to zero.

It is strongly advised that only the "B" should be specified in images used by single and extended precisions values. This is because floating point format handles values much differently than integer format. If a "B" is not specified alone, Comm-Stor will output the bytes as they are obtained internally, from left to right. A highly sophisticated knowledge of floating point formats is needed by the programmer in order to determine which bytes are required to represent the desired values. Such knowledge is beyond the scope of this manual and, therefore, will not be discussed herein.

4. READ # USING

The READ # USING command is an additional feature of Sykes BASIC which enables the programmer to obtain values from a disk file or external device according to a specified format. Consequently, this command should only be used if the exact format of the data is known.

For example, suppose a device attached to the modem port on the Comm-Stor unit sends data to Comm-Stor in the form of one space followed by three digits, such as:

...345 675 342 096 756 875 234 546 123 989 675...

We would not be able to obtain the data without a READ # USING command, since there are no delimiters embedded in the data to separate the different values from one another. To obtain these different values, various images can be used. Below is a list of READ # USING commands which will correctly obtain the different values above.

```
READ #2 USING"4#";A
READ #2 USING"+#.#";A
READ #2 USING"S3#";A
READ #2 USING"A(32)3#D";A
READ #2 USING"$-##";A
```

As you can see in the third example above, the format used in the image for inputting the data may be the same format as that of the existing data. The various options available for use in images do not necessarily mean the same when used in input and output formats. Table #4-6 shows the options used for inputting data which are not used for the same purpose when outputting data.

Option	Description
#	obtain one character
#.#	obtain three characters
T#.#	obtain three characters
+# or #+	obtain two characters
-# or #-	obtain two characters
E#	obtain five characters
$#.##	obtain five characters
#,###	obtain five characters
#I(AB)#	obtain four characters
S#	ignore one character and obtain next character
N#	same as above
B30#	obtain 30 characters no matter what they are
30#D	obtain 30 characters, but if the configured delimiter character is encountered, no more characters are assigned to the specific variable.
A(32)30#D	obtain 30 characters, but if a space is encountered, treat it as a delimiter for the specified variable only. (See Appendix C)

TABLE #4-6

The READ # USING command may also be used to avoid a hard-copy printout or display of the characters entered from a device. There are many applications for this. For instance, passwords or user identification numbers may be entered from a device, but the characters will not be echoed (displayed or printed) on that device. Therefore, if the hard-copy were misplaced, this information would remain confidential. This is done by specifying an "X" in the beginning of a field. The

characters subsequently entered for that field will not be displayed or printed. Below is an illustrative example:

```
100 PRINT #1;"PLEASE ENTER PASSWORD"
110 READ #1 USING"X10#";A$
```

The ten characters entered by the operator will not be displayed on the terminal, but will be assigned to A$.

Comm-Stor will always recognize the value of a field to be completed when the delimiter character is encountered unless the "B" option is specified in the image. If the B option is specified, the specified number of characters will be obtained, even if a control character is in the file. For example, if the data in the file is:

```
ABC[RETURN]CD6A
```

and

```
READ #4 USING"B6#";A$
```

is issued, the value of A$ will be "ABC[RETURN]CD". (The [RETURN] character is the default for the delimiter character.) If the "B" was not specified, such as:

```
READ #4 USING"6#";A$
```

the [RETURN] tells Comm-Stor that the value is complete; A$ will contain "ABC".

When the "E" option is used in an image, Comm-Stor will obtain four more characters from the file. Therefore, the image "E##" is equivalent to the image "6#". For example, if the data in the file is:

```
1.26E+02
```

then

```
READ #4 USING"E#.#";A
```

will cause A to be equal to 1.26E+02, or 126. Additionally, if the data is:

```
12647
```

and

```
READ #4 USING"E#";A
```

is executed, it will cause A to equal 12647 even though the data is not represented in exponential notation.

The "N" and "S" options cause one character, either a null or a space, (for each "N" or "S" specified) to be skipped over or ignored. For example, if the data in the file is:

```
ØØØ1(5 nulls)2
```

and

```
READ #4 USING"3S#;5N#";A,B
```

is issued, the "3S" will cause the three spaces to be ignored, 1 will be assigned to A, "5N" will cause the five nulls to be ignored and 2 will be assigned to B. If the same READ # USING command is issued but the data in the file is:

```
AB11CDE432
```

A and B will still be equal to 1 and 2, respectively.

If operating in random mode (see Chapter 5, The File Manager), a "P" can be specified. This will reposition the pointer to the specified character position within the specified record. For example, if the third record in a file contains:

```
TEST123
```

and

```
READ #4,3 USING"P(5)###";A
```

is issued, A will be equal to 123. The argument used with the "P" option may be a numeric constant or variable. The system must be in random mode to use this option.

As mentioned previously, the configured delimiter character can be acknowledged as such if the "D" option is specifed. For example, if the data in the file is:

ABC[RETURN]123

and

READ #4 USING"7#D";A$

is issued, A$ will contain "ABC". However, if:

READ #4 USING"A(50)7#D";A$

is issued on the same data, A$ will contain "ABC[RETURN]1". This is because the delimiter for that particular field was changed to the character "2".

The following is a list of examples of equivalent images for the READ # USING command:

Image	Equivalent Image
$#	##
3$3#	6#
3#,3#	7#
2(3#,)3#	11#
3#I(-)##I(-)4#	11#
##-	3#
-##	3#
4#	4#
+4#	5#
#.#	3#
$##.##	6#

If the options in the above list are used, the variable list in the READ # USING command cannot be the same as the variable list in the PRINT # USING command. This is because dollar signs, commas, inserted characters and trailing positive and negative signs are not numeric values. To solve this problem, two different techniques may be used.

The first technique allows the same image to be used by the READ # USING and PRINT # USING commands. However, each numeric value in the PRINT # USING command formatted with these options should be changed to a character variable in the variable list of the READ # USING command.

For example,

```
300 IMAGE3$#;2S#;2(3#,)3#
310 PRINT #4 USING300;1;2;1234
```

will enter:

ØØ$1Ø2ØØØØØØ1,234

into the file. To access this data,

```
500 READ #4 USING300;A,B,C
```

cannot be used; it will result in an "ERROR - TM" message. (See Appendix G.) To access the data without getting an error,

```
500 READ #4 USING300;A$,B,C$
```

must be used. A$ will contain "ØØ$1", B will equal 2 and C$ will contain "ØØØØØØØ1,234". This technique is implemented solely for the benefit of those programmers who wish to use the same images for PRINT #4 USING and READ # USING commands.

The second technique uses different images in READ # USING commands and PRINT # USING commands. Instead of the above READ # USING command, we could have accessed the data by:

READ #4 USING"19#";A$

This will assign the value of "ØØ$1Ø2 ØØØØØØ1,234" to A$. Additionally,

READ #4 USING"3S#;2S#;7N#;N3#";A,B,C,D

or

READ #4 USING"3S#;3#;8#;N3#";A,B,C,D

will cause A to be equal to 1, B to be equal to 2, C to be equal to 1 and D to be equal to 234. Consequently, if the following two commands are executed:

```
30 IMAGE3$#;2S#;2(3#,)3#
PRINT #4 USING30;A;B;C
```

followed by:

```
60 IMAGE3S#;3#;8#;S3#
READ #4 USING60;D,E,F,G
F=F*1000+G
```

the values of D and E will be equal to A and B when the READ # USING command is executed and F will be equal to the value of C when the next command is executed. There are many different applications of this technique. The programmer can best become familiar with the READ # USING command through experimentation.

Character string values sent to a disk file by a PRINT # USING command can be assessed by a READ # USING command using the same image and variable list. If an empty field is specified in the image, such as:

```
READ #4 USING"#;;#";A,B,C
```

a standard READ # (Chapter 7) will be issued for that particular field.

It should be noted that READ # USING commands can access data from disk files regardless of how the data was entered. The format of the data to be accessed should be closely scrutinized and experimented upon until a full understanding of these techniques is reached.

Many programmers wish to obtain a one-character response from a terminal. For example:

```
500 PRINT #1 ;"ARE YOU DONE?"
510 READ #1 USING"#";A$
520 PRINT #1;
```

will display:

```
ARE YOU DONE?
```

on the terminal. The first character entered will be obtained by statement 510 and a [RETURN] will be issued immediately to the terminal by statement 520.

Assume that data being entered from the terminal is sent to the modem by a BASIC program. Suppose this modem needs data sent at 128 characters at a time, treating all special characters as regular ASCII characters. The program below will accomplish this as follows:

```
200 READ #1 USING"B128#";A$
210 PRINT #2 ;A$
220 PRINT #1 ;"ARE YOU DONE?"
230 READ #1 USING"#";A$
240 PRINT #1;
250 IF A$="N"GOTO 200
```

This page intentionally blank.

CHAPTER 5

THE FILE MANAGER

CONTENTS

1. FILES

A. General

Data can be entered into a BASIC program by using the READ, DATA, and RESTORE commands (as the program is written) and by entering values from the port devices (while the program is executing) through the use of the INPUT and READ #(0-3) commands. These techniques prove to be inefficient when the amount of data increases beyond a few items, when the data must be changed, or when more than one program needs to use the same data. In order to improve operations, Sykes BASIC provides the programmer with facilities to define and manipulate data files.

A Sykes data file consists of a sequence of data items transmitted between a BASIC program and a diskette file or some external device, such as a modem or a terminal. Each file on a diskette is symbolically referenced by a file name which may contain up to twenty characters, optionally followed by a plus sign and an extension of up to twelve characters, depending upon the particular configuration of the diskette. Comm-Stor supports two types of disk file structures: FIXED LENGTH disk files and VARIABLE LENGTH disk files. (See the Comm-Stor IV Configuration Manual, Sykes Publication No. 9990B0259 and Comm-Stor IV Reference Manual, Sykes Publication No. 9990B0258).

Sykes File Manager Package contains a set of BASIC commands which perform certain operations on disk files. These commands are:

APPEND
ASSIGN
CALL
CLOSE
CREATE
IF END #
KILL
LINK
LOAD
MERGE
OPEN
POSITION
PRESS
PRINT #
PRINT # USING
READ #
READ # USING
RECSIZE
RELEASE
REWIND
SAVE
TRUNCATE
UNLOCK

A section on each diskette (located at the first sector of Track 1), called the diskette header (Figure #1-5), is produced at configuration time and contains information about each diskette. When a CREATE command is executed, Comm-Stor associates a certain portion of the diskette, called a FILE SLOT, with the specified file name. Comm-Stor then adds information about the disk file into the corresponding Directory

entry (Appendix F) located in the diskette Directory. The CREATE command acts differently according to which disk file structure is used.

B. Fixed Length Files

Because every file on a fixed length diskette is the same size,

CREATE<file name>

is the format of the CREATE command that must be used. If the command,

CREATE"TEST"

is executed, Comm-Stor determines if an empty file slot (the size is determined at configuration time) is available. If one is available, the disk header is updated, a Directory entry is formed, and the BASIC program will continue. If an empty file slot is not available, an "ERROR-FULL DISK" (Appendix G) will result.

C. Variable Length Files

Variable length files provide the programmer with a greater amount of flexibility regarding the allocation of disk file space. If the files residing on a diskette are not the same size, greater utilization of diskette space is obtained when a large number of files is specified at configuration time. For example, if 100 files are specified at configuration time, then the disk may contain from one up to 100 files, depending on the size of each file slot.

At configuration time, the <u>average</u> file size must be calculated and stored in the disk header by Comm-Stor.

Using one of the following commands, the program can determine if an average size, maximum size, or specified size file slot is available. If

CREATE"TEST"

is executed, Comm-Stor determines if an empty file slot (of average file size) is available. The largest (maximum) file slot can be located by entering:

CREATE"TEST",M

To find an available file slot containing a specified number of sectors, the programmer might for example enter the following:

CREATE"TEST",67*128

Comm-Stor will find an available file slot of 67 sectors (1 sector = 128 characters).

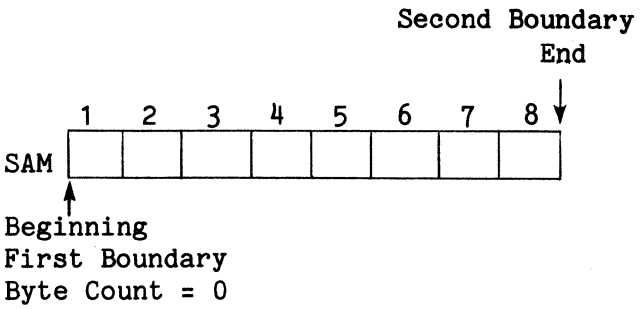
2. DISK FILE ACCESS

Once a disk file is CREATEd, it may be accessed in one of two ways: either SEQUENTIALLY or RANDOMLY.

As an example, let us CREATE a file called "SAM" with 8 sectors of disk space. We will execute the following command:

CREATE"SAM",8*128

We can represent what has happened on the disk as follows:



As a result of this command, note that nothing has been entered into the file but something already may be there from some previous disk operation. The file was not related to a BASIC PROGRAM, nor was it used in any active way. Disk space was allocated for 8 contiguous sectors and information was added to the directory entry showing the beginning, ending, and current byte count (which is the total number of characters contained in the file - 0 from a CREATE).

Parameter	Default Value	Requirements
File Reference Number	Specified by the user	Must be a whole number between 4 and 24, inclusive.
Drive Number	1	1 is for upper drive 2 is for lower drive
File Name	Specified by the user	Determined at configuration time
Record Size	128 characters	1 character to total file size
Output to a Protected File	NO	--
Buffer Size	1 sector	Depends on size of user memory*
Double Buffering	NO	Depends on size of user memory*

*Appendix B explains the allocation of buffers in the Comm-Stor IV system.

TABLE #5-1

A. The OPEN Command

We would now like to enter data into disk file "SAM". First, the disk file must be related to the BASIC program, and parameters must be specified concerning how the file is to be accessed. The OPEN command is used for this purpose (page 7-30).

The OPEN command requires seven different parameters. Five of the parameters have default values since they are used quite frequently. Therefore, these five parameters will not be included in the OPEN command. The seven parameters and their default values, where applicable, are listed in Table #5-1.

It can be seen from the table above that only two parameters are required: the FILE REFERENCE NUMBER and the FILE NAME. All others will default to the given values if they are not specified in the OPEN command. For our example, let us execute:

```
OPEN#4TO"SAM"&3
```

This command relates the disk file (on the upper drive of a dual drive unit) to reference number 4. Since no record size is specified, it defaults to 128 characters. The buffer size is three sectors, and only one buffer is allocated to the file reference number. Additionally, a file header is stored in the File Buffer with the specified buffer space. This file header contains information about the parameters specified for the file reference number. With this background, we can now discuss the two ways of accessing disk files.

B. Sequential Access

Comm-Stor keeps track of the boundaries of a file accessible by the programmer. When accessing disk files sequentially, the boundaries are the first and last character positions of the file. Data may be entered into the file starting at the first boundary, followed consecutively by additional data, until the

second boundary is encountered. For example, if we entered 130 characters into file "SAM" by the command sequence:

```
FOR I=1 TO 130
A$ = A$+"A"
NEXT I
PRINT #4;A$
```

The file will now look like Figure #5-1.

Notice the 130 "A"s were entered consecutively from the first boundary. Additionally, the delimiter was entered into the file directly after the data. The delimiter is always entered into a disk file after a value is entered in order to distinguish that value from others. Consequently, the total number of characters, or byte count, for the file is 131 (130+1=131). Also notice that the first boundary marker called the pointer (^), is now at the 132nd position in the file.

Entering:

```
PRINT#4;"A";"TEST"
```

will now cause the file to look like Figure #5-2 and the first boundary marker is changed to the current position of the pointer, which is the 139th character position in the file.

We can keep entering data into the file "SAM" in the same manner, provided that we do not try to exceed the second boundary. If we do, an "ERROR - EF" (End of File) will occur.

A special feature of the Sykes File Manager Package is the ability to automatically fill a file between the first and second boundaries with any character the programmer chooses. This is done by executing:

```
PRINT #4; FILL(32)
```

where 32 is the decimal value for an ASCII space (Appendix C). The remainder of

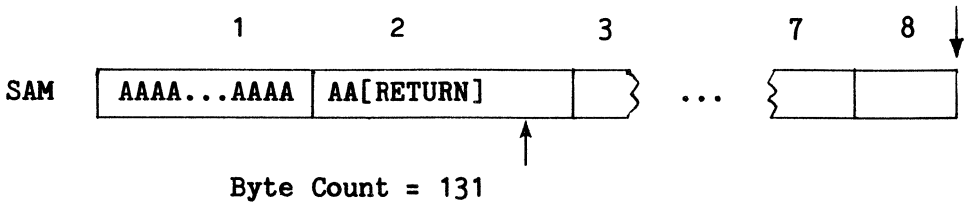


FIGURE #5-1

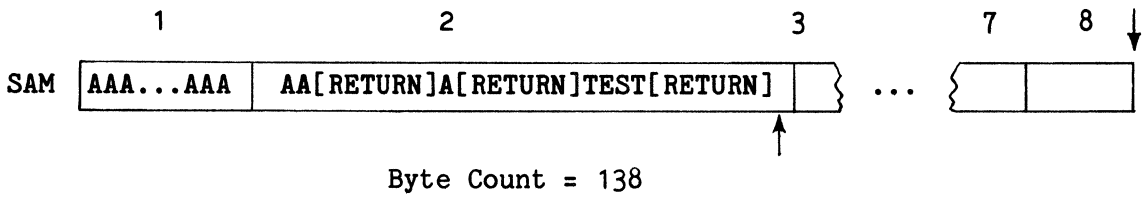


FIGURE #5-2

"SAM" will now be filled with spaces, and the file will appear as illustrated in Figure #5-3, where the character "Ø" signifies a space. This is a convenient method of removing any data which might exist in the file from some previous disk file operation.

Suppose that we would now like to obtain the data which we just entered into "SAM". Since the first boundary is now at the end of the file, we have to move it back to the beginning of the file in order to access the data.

To do this, execute:

```
REWIND #4
```

This command will set the first boundary marker to the beginning of the file.

Now execute:

```
READ #4;A$
```

The pointer will point to each successive character position, and Comm-Stor will assign these characters to A$. When the pointer points to the delimiter, the system assumes the value of the variable to be complete. Therefore, A$ contains 130 "A"s. If we now execute:

```
READ #4;B$,C$
```

B$ will contain "A" and C$ will contain "TEST".

The pointer, once again, is at the 139th character position in the file.

C. Random Access

To obtain the second value entered into "SAM", we could REWIND the file, READ the first value, and then READ the second value. However, it would be much easier if we could simply position the pointer to the beginning of the second value. We can do this by executing:

```
READ #4,2 USING"P(4)";A$
```

This is a <u>random access</u> command. It positions the pointer to the beginning of the

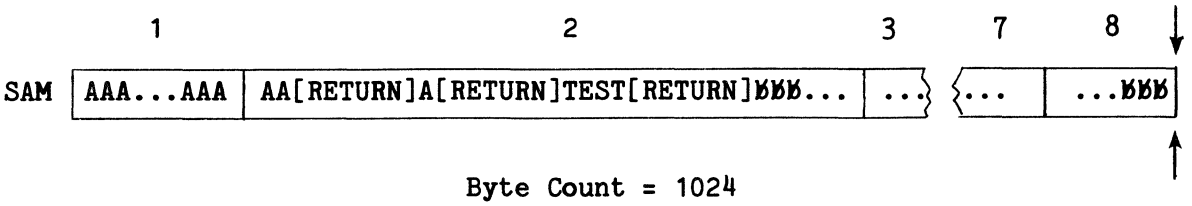


FIGURE #5-3

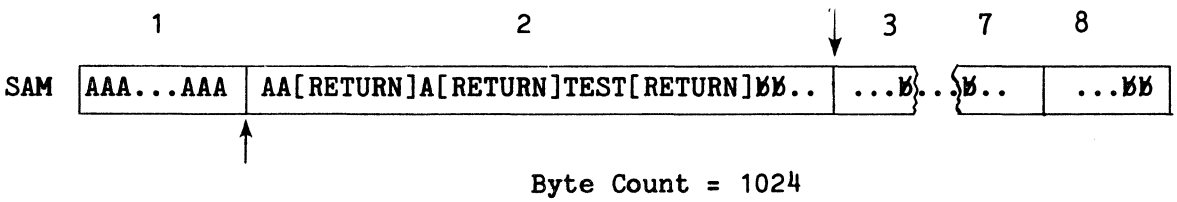


FIGURE #5-4

second record in the file; the boundaries for sequential access were changed to the beginning and ending of the second record. (Remember, the record size defaulted to 128 characters in the OPEN command.) The file then looks like Figure #5-4.

The "P" option in the image repositioned the pointer to the 4th character position from the first boundary and obtained the value of A$. The file now looks like the example in Figure #5-5, and A$ contains "A".

Comm-Stor always accesses data sequentially until the boundaries are changed by the programmer. Therefore, when the second record was specified in the above command, the boundaries were changed and, consequently, random access of the data occurred. Additionally, the "P" option was specified in the image. This, too, is random access because the programmer changed the first boundary to the specified position in the record.

The value for A$ was obtained sequentially, however. If we execute:

```
READ #4;A$
```

another sequential access of the data is performed, and A$ contains "TEST". "SAM" now appears in Figure #5-6. Note that the first boundary marker is now at the 11th character position in the second record of the file. If we try to execute:

```
A$=""
FOR I=1 TO 128
A$ = A$+"B"
NEXT I
PRINT #4;A$
```

We will get an "ERROR - EF" because the data would exceed the second boundary. We can, however, move the second boundary to the end of the file by executing:

```
UNLOCK #4
```

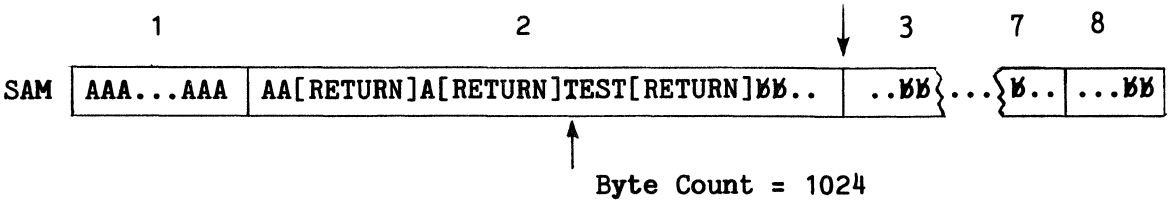


FIGURE #5-5

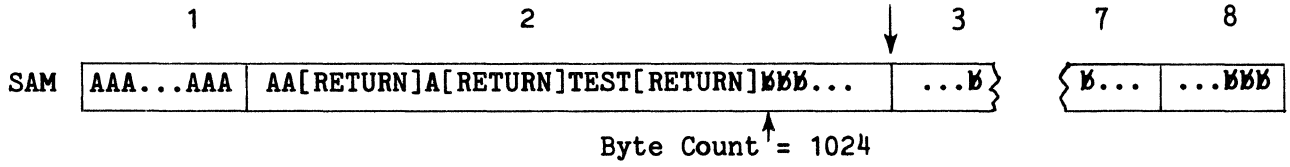


FIGURE #5-6

This command keeps the first boundary stationary. If we now executed:

PRINT #4;A$

the file will appear as illustrated in Figure #5-7.

D. The CLOSE and RELEASE Commands

When operations with a file are completed, Comm-Stor must be advised. We can use one of two different commands to accomplish this: CLOSE or RELEASE.

A CLOSE command tells Comm-Stor that the file associated with the file reference number should no longer be related to that reference number. We could execute:

CLOSE #4

This command would disassociate the file "SAM" from the BASIC program. The last buffer is written to disk if a WRITE operation had been done, and the byte count in the Directory entry is updated to reflect the new byte count in the file. The buffer space, 3 sectors plus the space used by the file header, is still associated with that file reference number.

Instead of the CLOSE command, we could execute:

RELEASE #4

Like the CLOSE command, this command also tells Comm-Stor that the file is no longer related to the reference number. This command also tells Comm-Stor that the File Buffer space associated with the file reference number is no longer related to that reference number. Since we RELEASEd File Buffer space, we may now use this space for other file reference numbers. (If another file reference number is used, it will use this same RELEASEd File Buffer space.)

E. The ASSIGN Command

The ASSIGN command is used to associate a disk file with a file reference number after File Buffer space has been permanently set (possibly by previous use of a character variable) and to use any available File Buffer space for that reference number.

The first OPEN command allocated three sectors plus room for one file header for the BASIC program's File Buffer space. We then RELEASEd this space making it available for use by other file reference numbers. Since the smallest amount of File Buffer space that may be used by a file reference number is one sector plus the file header, three file reference numbers cannot be used. This would cause an "ERROR - FO" as the space for three sectors plus three headers is larger than the available File Buffer space. Therefore, for this example, we will use two file reference numbers.

Suppose that a disk file presently exists which is called "PRODUCTS" and contains 2,000 records. Each record has a maximum length of 51 characters. The fields in each record are:

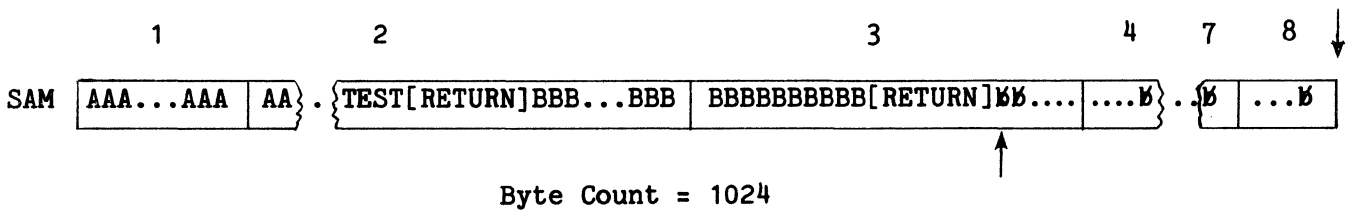


FIGURE #5-7

PRODUCTS

Field	Description	Characteristics
1	Product code	1-10 characters
2	Product name	1-20 characters
3	Quantity on hand	1-6 digits
4	Unit price	1-7 digits (XXXXX.XX)
5	Last sale date	8 characters (MM-DD-YY)

Using the limit of two file reference numbers, we can easily update the "Quantity on hand" field of each record, find the total number of products in stock, and add this information to another existing file called "TOTALS" which will be used for generating a report. We can now:

```
ASSIGN #5 TO "PRODUCTS",51
```

which relates file reference number 5 to the disk file "PRODUCTS", using one sector plus 35 bytes for a file header, with a record length of 51 characters.

We may also:

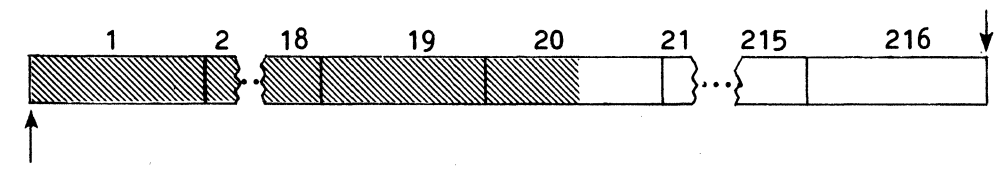
```
ASSIGN #6 TO "TOTALS"
```

which relates file reference number 6 to the disk file "TOTALS". We now have 93 bytes of available space left in the File Buffer (128*3+35 - (128+35) - (128+35) = 93). File reference numbers 5 and 6 each use 163 bytes of File Buffer space. We can now execute the following BASIC lines:

```
APPEND #6
INPUT "WHAT IS THE DATE, IN THIS FORM:
    MM-DD-YY";D$
PRINT #6 USING "7#;8#D";"DATE:",D$
FOR I=1 TO 2000
POSITION #5,I
READ #5 USING"10#";PC$
PRINT "PRODUCT CODE =";PC$
INPUT "NUMBER SOLD";AMT
READ #5,IUSING "P(31)6#";QOH
QOH = QOH - AMT
TOT = TOT + QOH
IF AMT <>0 THEN PRINT #5,IUSING "P(31)6#;
    P(44)8#";QOH,D$
PRINT #6 USING "10#;5S6#D";PC$,AMT
NEXT I
PRINT #6USING "8#;#,3#,3#,3#D"; "TOTAL =",
    TOT
END
```

Notice that there is not a CLOSE command in the above group of lines. This is because all active file reference numbers are automatically CLOSEd when a BASIC program completes execution.

The APPEND command is used to position the pointer to that character position in the disk file immediately to the right of the last character entered in the file. For example, suppose the file "TO-



Byte Count = 2550

FIGURE #5-8

TALS" looked like Figure #5-8 after we executed the ASSIGN command for file reference number 6.

The "cross-hatching" signifies data stored in the file. After the APPEND command is executed, the file will look like Figure #5-9, and the pointer is at the 2551st character position in the disk file. After the date is obtained from the terminal (for example, "06-22-78"), it is entered into the file, which will now appear as in Figure #5-10.

Assuming that when we executed the ASSIGN command for file reference number 5 the disk file look as in Figure #5-11, where the numbers 1 through 2000 refer to the record numbers and not the sectors.

The first POSITION command then causes the file to look like the Figure #5-12.

Let us also assume that the first record contains the following values:

Value	Field
"0102-ABC-3"	Product code
"MUSHROOM SOUP"	Product name
21567	Quantity on hand
0.43	Unit price
"05-25-78"	Last sale date

When the next READ #5 USING command is executed, the first ten characters are obtained from this record and assigned to the variable PC$. The disk file now appears as illustrated in Figure #5-13, and PC$ contains the characters "0102-ABC-3".

This "Product code" is displayed on the terminal and the operator enters the corresponding number of items sold; this number is assigned to the variable AMT. The next READ #5,IUSING command repositions the pointer to the 31st character position in the record; six digits are obtained from the file and assigned to the variable QOH. The disk file now looks like Figure #5-14, and the variable QOH is equal to 21567.

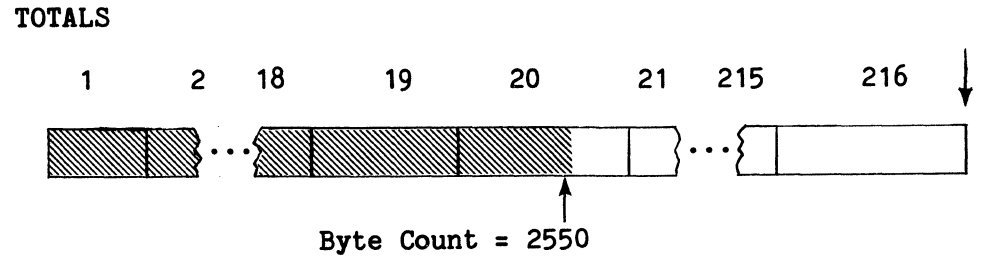


FIGURE #5-9

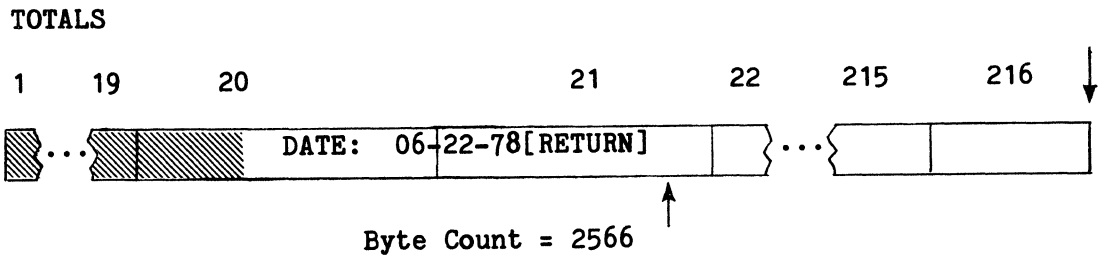
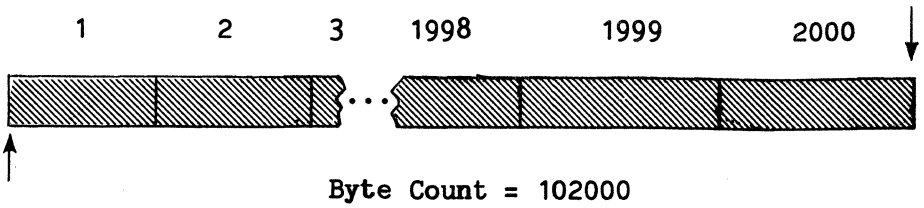


FIGURE #5-10

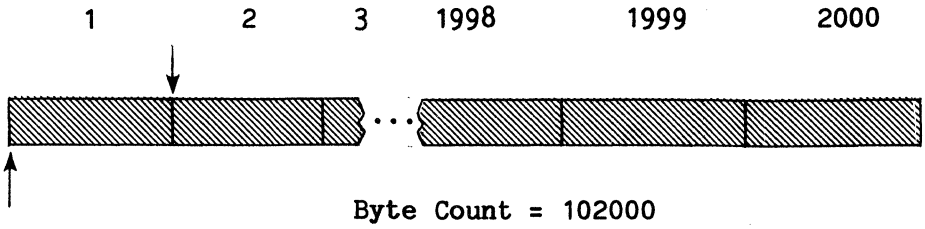
PRODUCTS



Byte Count = 102000

FIGURE #5-11

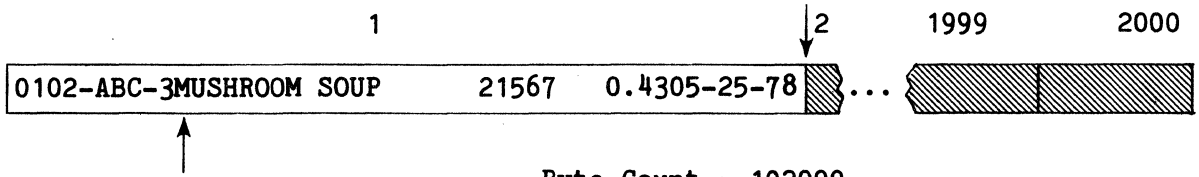
PRODUCTS



Byte Count = 102000

FIGURE #5-12

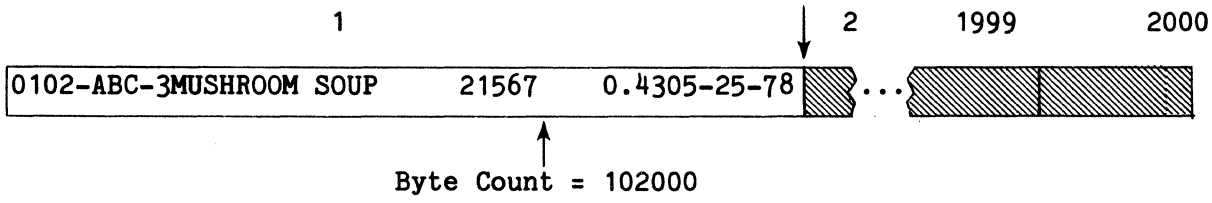
PRODUCTS



Byte Count = 102000

FIGURE #5-13

PRODUCTS



Byte Count = 102000

FIGURE #5-14

The number of items sold (AMT) is then subtracted from this number to find the number of items left in stock. This result is then added to the total number of items in stock (TOT). If the number of items sold for this product equals zero, there is no change to the record, and the file is not changed. Assuming the operator entered the value 253 into the variable AMT, QOH now would equal 21314 (21567 - 253 = 21314). The PRINT #5 command then repositions the pointer back to the 31st character position, enters the value of QOH, repositions the pointer to the 44th character position, and enters the contents of D$. Only these fields will be changed; the other part of the record is not changed in any way by using the "P" option in the image. Therefore, the disk file now appears as shown in Figure #5-15.

The "Product code" and number of items sold for the product are then entered into the "TOTALS" disk file which now looks like Figure #5-16.

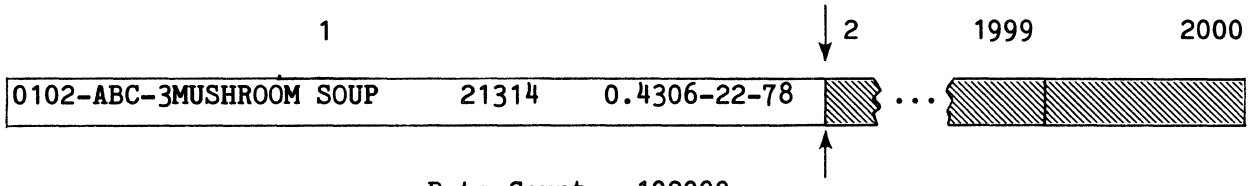
These operations are then performed for the other 1999 records which exist in the disk file "PRODUCTS". When these records are completely processed, the last PRINT #6 USING command will be executed, entering the total number of items sold on this date. The disk file "TOTALS" is now shown in Figure #5-17 and processing for this example is now complete.

If a COM ".D TOTALS" is now executed, the portion of the file just entered will look like:

```
DATE:  06-22-78
0102-ABC-3      253
    .            .
    .            .   (other 1999 products)
    .            .
TOTAL = 226,789,431
```

This completes the discussion of this example.

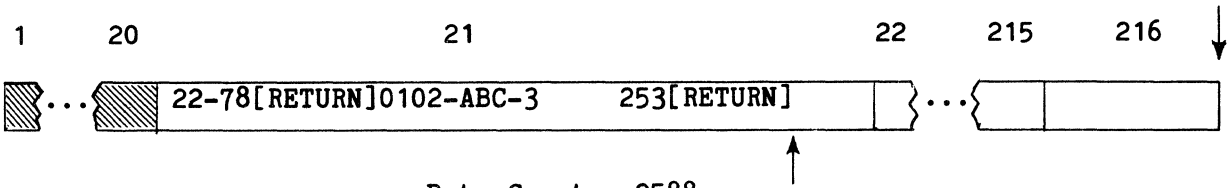
PRODUCTS



Byte Count = 102000

FIGURE #5-15

TOTALS



Byte Count = 2588

FIGURE #5-16

F. The RECSIZE Command

Another command is used in the File Manager Package to instruct Comm-Stor to change the record size parameter in the corresponding file header. For example,

```
OPEN #5 TO "TEST"
```

causes the record size to be 128 characters long. To change the size of each record in "TEST" from the default value of 128 characters to a specified value, the RECSIZE command is used. Entering:

```
RECSIZE #5 TO 80
```

changes the record size from 128 characters to 80 characters. If this command were not available, the programmer would need to first CLOSE #5 and then ASSIGN it with a new record size.

G. Functions

There are two functions in Sykes BASIC, POS and FILL, which may be used with the File Manager Package. When the file reference Number is used with the POS function, the pointer is returned to the current position in the file. If a file has been accessed sequentially, the result of POS will be the last accessed character position in relation to the beginning of the file. (If the file has not been accessed, the result of POS will be 1.) If the file has been accessed randomly, the result of POS will be the current record number. For example, using the file reference number 5 that has a record size one sector long and has been accessed sequentially, the pointer is positioned at the first position in the second record.

```
A = POS(5)
```

will assign the value of 128 to the variable A. If we had positioned the first boundary to the second record, however, A would be assigned the value 2.

As described on page 5-4, the FILL function is used to output a character to the file repeatedly until the second boundary is encountered. The argument of the FILL function is a decimal number which is equivalent to an ASCII character (obtained from Appendix C). For example, if a file has been accessed sequentially (in this case, through file reference number 6) and:

```
PRINT #6;A;FILL(32)
```

is executed, the value of A would be output to the file and the remainder of the file will consist of spaces. However, if:

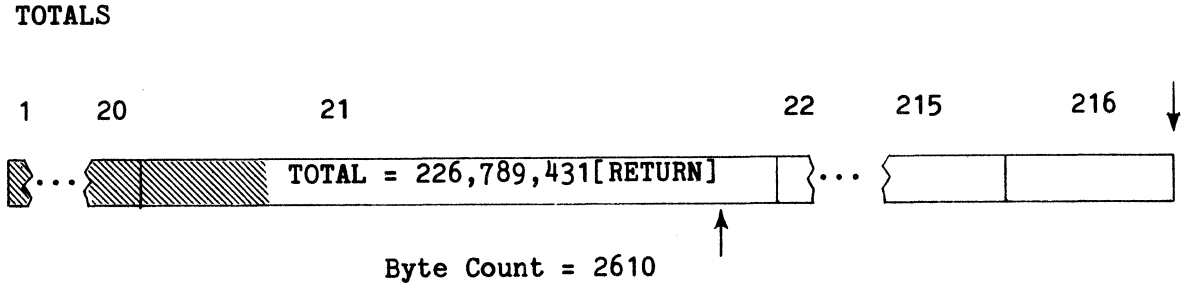


FIGURE #5-17

```
POSITION #6,2
PRINT #6;A;FILL(32)
```

is executed, the value of A will be output to the second record of the file and the remainder of the record will consist of spaces.

3. EXTERNAL DEVICES

The only external devices which can be attached to the Comm-Stor unit are a modem, terminal, and printer. Data is transferred to and from these devices through a port reference number.

Port Reference Number	Device
0	invoker
1	terminal
2	modem
3	printer

The invoker is the device which issues the ".BA" command to Comm-Stor.

As described in Appendix B, the buffer sizes for the terminal and modem devices are specified at configuration time; the size of the modem buffer may be changed during a BASIC program using the OPEN command. These buffers are used by Comm-Stor to temporarily store incoming data. When data is output, no buffers are used since the data is transferred one character at a time to the device.

Data is transferred to and from these devices by the following commands:

```
PRINT #
PRINT # USING
READ #
READ # USING
PRINT          )
PRINT USING    } (invoker only)
INPUT          )
```

Data from these devices can only be accessed sequentially. Additionally, the POS function can be used for the modem, terminal, or invoker's buffer by passing the port reference number as the argument.

4. MULTIPLE REFERENCE NUMBERS

When a PRINT # or PRINT # USING command is issued, more than one reference number may be specified. For example, to enter the characters "TEST" into a file associated with file reference number 5 and output the data to the printer, modem, and terminal, we could execute:

```
PRINT #5&#1&#2&#3;"TEST";
```

The reference numbers must appear in a certain order in the command. The rules for entering and outputting data are determined by the first reference number specified. For example, a disk file reference number is specified first, so the delimiter will be output after "TEST". However, if the above command was changed to:

```
PRINT #1&#5&#2&#3;"TEST";
```

the delimiter would not be output.

5. END OF FILE DETECTION

As was described on page 3-14, the programmer can control the procedure in a BASIC program when an error condition arises using the IF ERROR # command. For example, if the command:

```
IF ERROR #27 GOTO 5000
```

is specified in a BASIC program when an end of any file is encountered, program control will be transferred to line 5000.

If the programmer is using more than one file at a time, this command will not enable the programmer to know in which file the error was detected. Therefore, the File Manager Package supplies the command, IF END #, which allows the programmer to specify a particular file reference number.

For instance, if the commands:

```
IF END #4 GOTO 5000
IF END #5 GOTO 5500
```

are executed, program control will be transferred to a particular line in the BASIC program depending upon the file in which the error occurred.

6. VIRTUAL ARRAYS

A special feature of Sykes BASIC is the implementation of virtual arrays. Virtual arrays are the same as regular arrays (page 2-9) except that:

1.) the values for the array's elements are automatically obtained by Comm-Stor,

2.) a disk file contains the current values of the array, and

3.) a total of 11 bytes, plus the number of bytes determined by multiplying the number of dimensions by 2, is allocated in internal memory for each array.

As an example, suppose a disk file named "INVENTORY" resides on the disk inserted in the drive (the upper drive of a dual drive unit). "INVENTORY" contains 200 values; the first 100 values are the product numbers and the last 100 values are the quantities in stock. If virtual arrays are not used, the following commands would have to be executed to obtain the values of the array:

Example I:

```
OPEN #4 TO "INVENTORY"
DIM INV(99,1)
FOR J=0 to 1
FOR I=0 to 99
READ #4;INV(I,J)
NEXT I
NEXT J
```

Virtual arrays operate more efficiently and simplify programming. To obtain the values for the virtual array, INV, from file "INVENTORY", only two commands are required:

Example II:

```
OPEN #4 TO "INVENTORY"
DIM #4 TO INV(99,1)
```

Each element of array, INV, is now assigned a value from the "INVENTORY". It should be noted that the order in which the two commands above are executed does not matter.

The OPEN or ASSIGN statement for virtual arrays may specify a buffer size if desired. If a record size is issued in this command, it will be ignored since the virtual array specifies its own record size.

More than one virtual array may be associated with a file by issuing:

```
DIM #4 TO PROD(99), QUANT(99)
```

instead of the previous DIM command.

CHARACTER ARRAYS may also be used as virtual arrays. 32 characters in the file are automatically assigned to each element. For example:

```
OPEN #5 TO "TEST"
DIM #5 TO C$(5,2)
```

will cause C$(0,0) to contain the first 32 characters in "TEST", C$(1,0) will contain the next 32 characters, and so on. If the programmer wishes to have 50 characters assigned to each element,

```
DIM #5 TO C$(5,2)=50
```

may be issued. Remember, however, that a character string cannot contain more than 255 characters.

Another important function of virtual arrays is to change the value of an element or elements. If we had a program containing the commands in Example I and at some later time we executed:

```
INV(60,1)=A/B
```

the value of INV(60,1) would be changed in internal memory; the disk file "INVENTORY" would not be changed in any way. However, if the program contained the commands in Example II and we executed the command above, the value in the disk file "INVENTORY" which is associated with INV(60,1) will now be changed to the value of A divided by B. This occurs because each element of a virtual array is an index to a disk file; a table is not stored in internal memory for each virtual array. This minimizes program execution time and simplifies updating stored data.

The virtual arrays are always associated with the file reference number specified in the DIM command unless an ERASE command is executed. For example:

```
 10 OPEN #4 TO "TEST1"
 20 DIM #4 TO B(250,5),D$(10,10)=40
  .
  .
  .
 90 CLOSE #4
100 ASSIGN #4 TO "TEST2"
  .
  .
  .
190 CLOSE #4
200 ERASE B,D$
210 ASSIGN #4 TO "TEST3"
  .
  .
  .
```

The values of virtual arrays B and D$ are first obtained from disk file "TEST1" in line 20. When line 100 is executed, the data in file "TEST2" is assigned to the virtual arrays. When line 200 is executed, the use of virtual arrays would no longer be possible (Chapter 7, ERASE); and consequently, when line 210 is executed, no virtual arrays will be associated with file "TEST3". However, if line 200 is changed to:

```
200 ERASE B
```

then virtual array D$ will be associated with file "TEST3" when line 210 is executed.

Disk files associated with virtual arrays may be treated as regular files. That is, PRINT commands may change the contents of the files but will also change the affected element(s) of the virtual array(s).

Virtual arrays may also be associated with different files in one DIM command. For example:

```
DIM #4 TO B(10,10);#5 TO C$(2,2),F(5)
```

will associate virtual array B with file reference number 4 and virtual arrays C$ and F with file reference number 5. Additionally, a regular array may be defined in the same DIM command as a virtual array, such as:

```
DIM #4 TO B(10,10);#5 TO C$(2,2),F(5);D
(5,5)
```

Array D is not a virtual array. Caution is advised when creating a disk file to be used with virtual arrays. As with any newly created file, it contains meaningless characters. Therefore, it is suggested to fill newly created files with zeros or spaces so that the virtual arrays will not contain these erroneous values.

Additionally, the size of a disk file must be equal to or greater than the number of storage bytes used by the virtual array(s). To calculate the number of bytes needed to store a virtual array, the following table may be helpful:

Element type	Number of storage bytes per element
integer	3
single precision	5
extended precision	8
character	1 plus the length of each element

TABLE #5-2

Determine the number of bytes needed for each virtual array; the result will be

the smallest number of bytes possible to create a disk file which will be associated with the virtual array(s). For example, if we wanted to use the integer virtual array A%(19), we would determine that 3 bytes multiplied by 20 elements equals 60 bytes. Therefore, at least 60 bytes must be in the corresponding disk file.

If the disk file is not large enough, an End of File condition ("ERROR - EF") will occur when referencing an element outside the bounds of the file. If the virtual array(s) reference an address outside the bounds of the entire disk, a diskette error ("ERROR - DISKETTE") will occur. A good way to ensure that the virtual array elements do not overflow the boundary is to reference the last element before performing calculations. If an error does not occur, you know that you have enough disk space associated with the virtual array(s) used.

As a final note, the OPTION BASE command (Chapters 2 and 7), if executed, will also affect virtual arrays.

CHAPTER 6

THE REAL-TIME CONTROL SYSTEM

CONTENTS PAGE

1. GENERAL

The Real-Time Control System (RTCS) is a powerful tool used to detect external conditions from within a BASIC program. These external conditions are often called INTERRUPTS since they cause a break in the continuity of a BASIC program when detected. The different interrupts that can be detected by the programmer are classified by seven major interrupt categories. Each category has a reference number associated with it, as shown below:

Reference Number	Interrupt
1	TERMINAL
2	MODEM
3	PRINTER
4	TIMER 1
5	TIMER 2
6	REGISTER 1
7	REGISTER 2

Usually, when a BASIC program is executing, and does not request any data from a device by a READ or INPUT command, any characters entered from a device will be stored in the buffer for that device. For instance, a character sent to the modem port will be stored in the Modem Buffer and a character entered from the terminal will be stored in the Terminal Buffer.

When the program does request input from a device, Comm-Stor first checks to see if there is data in the corresponding buffer. If there is not, Comm-Stor will delay execution of the program until data is entered from the specified device. Consequently, time can be wasted if there is not any input for a period of time; the program could possibly be performing other operations until interrupted and told that data is ready to be input. Without using the Real-Time Control System, a [CNTRL/T] could be issued, data could be entered, and the program could then continue provided that no errors occur when this is done. Utilizing the Real-Time Control System, however, provides the programmer with the security that all operations will be carried out at the most efficient level.

A program can be prepared to accept and deal with any interrupts that might occur by using the WHEN command. The command format looks like:

WHEN #<ref> { GOTO / GOSUB / END } <line num> [{ &<priority>[,E] / &E }]

Substituting in the corresponding reference number and the desired line number, the program will then be able to acknowledge the interrupts and take the appropriate actions. For example, these commands:

```
WHEN #2 GOTO 5000
WHEN #4 GOSUB 6000
```

will cause program control to be transferred to line 5000 if a MODEM interrupt occurs and to execute the subroutine starting at line 6000 whenever a TIMER 1 interrupt occurs.

Example: Company XYZ processes the payroll for each of its employees on a Comm-Stor unit. Because the employees get paid only once a month, XYZ offers a special service to them; any employee can enter his/her employee number at the terminal while the payroll program is executing and can immediately obtain his/her current account status. This is done by the following program:

```
        10 WHEN #1 GOSUB 5000
        20 RTCS 1
        30 OPEN #5 TO "PAYROLL"
Payroll .
Program .
        .
        4000 END
        5000 RTCS 0
        5001 READ #1; EMPNUM
        5010 READ #5,EMPNUM;A$
        5020 PRINT "YOUR CURRENT STATUS
             IS:"
        5030 PRINT A$
        5040 RTCS 1
        5050 RETURN
```

Line 10 tells Comm-Stor to execute the subroutine starting at line 5000 if a TERMINAL interrupt occurs. Line 20 enables interrupts, or activates the Real-Time Control System. Therefore, the main body of the payroll program can be executing and, if an employee number is entered from the terminal, the appropriate record of the payroll file will be displayed. Upon execution of the RETURN command, the program will then continue where it left off before the interrupt occurred.

The format of the RTCS command is:

RTCS <log exp>

Whenever the logical expression produces a nonzero (true) result, all interrupts specified by WHEN commands will be enabled.

All interrupts specified by WHEN commands are checked after one BASIC line has been executed and before the next BASIC line is executed. As more than one interrupt may occur at the same time (such as a character being entered through the terminal port at the same time a TIMER 2 interrupt occurs), priorities can be specified in WHEN commands. Since there are seven different interrupt categories, there are seven priority levels. The commands below:

```
WHEN #1 GOSUB 6550&1
WHEN #2 GOTO 7000&3
WHEN #6 GOTO 596&6
WHEN #4 GOSUB 5000&7
WHEN #3 GOSUB 7500&5
```

will cause the priority of BASIC operations, from first to last, to be:

```
TIMER 1
REGISTER 1
PRINTER
MODEM
TERMINAL
```

Therefore if the interrupt routine caused by REGISTER 1 is executing (Priority 6), only a TIMER 1 interrupt (Priority 7) could interrupt it.

The detection of an interrupt can be stopped when necessary. For example, suppose a certain program will execute for two days or longer. Suppose, too, that we only want to perform a BASIC operation the first time a TIMER 1 interrupt occurs. By specifying:

```
WHEN #4 END
```

after the subroutine starting at line 5000 has been completed, Comm-Stor will no longer check the TIMER 1 interrupt. It should be noted here that when a [CNTRL/T], a [CNTRL/S], or a [CNTRL/Q] is entered through the terminal port, it is considered a special character in this case. These characters will not be considered as a TERMINAL interrupt for the Real-Time Control System; they will be operated on in their normal fashion, as described in Chapter 2.

2. THE TIMERS

In order to use TIMER 1 or TIMER 2, Comm-Stor must first know the actual time. The SETIME command is used for this purpose. When a specific time is recorded into Comm-Stor, it must be a character string (containing 6 numeric characters) and must be entered according to a 24-hour clock, where the last two digits represent seconds. Midnight is considered the "zero" hour and is denoted by "000000".

For example, suppose it is 5 seconds past 2:18 P.M. We can enter this actual time into Comm-Stor by executing:

```
SETIME #0 TO "141805"
```

Suppose, too, that at 5:00 P.M. and at 10:15 P.M. we would like to perform some BASIC operations. We can enter these two times into Comm-Stor by executing:

```
SETIME #1 TO "170000"
SETIME #2 TO "221500"
```

These two commands will enter 5:00 P.M. into TIMER 1 and 10:15 P.M. into TIMER 2.

After the TIMERs are established, the respective WHEN commands must be executed for this example by executing:

```
WHEN #4 GOTO 5000
WHEN #5 GOTO 6000
```

a TIMER 1 interrupt will occur at 5:00 P.M., a TIMER 2 interrupt will occur at 10:15 P.M., and the appropriate BASIC operations will be executed.

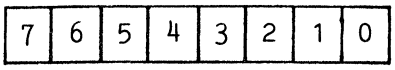
3. THE REGISTERS

There are two hardware registers that may be used by the programmer in the Real-Time Control System. To use these registers, the WATCH commands must be executed. The format for these commands is:

WATCH { 1 / 2 } <address>,<val 1>[,<val 2>]

The address signifies a location in memory which the programmer associates with a hardware register. Each location consists of one byte and contains 8 bits. Each bit represents a condition which can be detected by the programmer. The first value specified is called the "AND value" and the second value specified is called the "TEST value". (If the test value is not specified, it is assumed to be 0.) Listed below are the values of the conditions and the location and bit where each of them can be found.

BIT LOCATIONS ON ONE BYTE



Value	Location	Bit
"Diskette 1 READY" READY = 0 not READY = 1	16	6
"Diskette 2 READY" READY = 0 not READY = 1	16	5
"Request to Send" line from the printer Asserted = 0 not Asserted = 1	9	2
"Sec. Request to Send" line from the printer Asserted = 0 not Asserted = 1	9	1
"Data Terminal Ready" line from the printer Asserted = 0 not Asserted = 1	9	0
"Clear to Send" line from the modem Asserted = 0 not Asserted = 1	3	4
"Data Set Ready" line from the modem Asserted = 0 not Asserted = 1	3	3
"Carrier Detect" line from the modem Asserted = 0 not Asserted = 1	3	2
"Secondary Carrier Detect" line from the modem Asserted = 0 not Asserted = 1	3	1
"Ring" line from the modem Asserted = 0 not Asserted = 1	3	0

Value	Location	Bit
Binary Switch Setting Enter mode = 0 Exit mode = 1	1	4
"Request to Send" line from the terminal Asserted = 0 not Asserted = 1	1	2
"Sec. Request to Send" from the terminal Asserted = 0 not Asserted = 1	1	1
"Data Terminal Ready" from the terminal Asserted = 0 not Asserted = 1	1	0

This list is illustrated on the following page.

Because all values are stored internally as binary values, the values specified in the WATCH commands should be decimal values which represent the bit positions in the desired location. For example, to detect a change in the binary switch setting, the AND value would be 16. This is because that condition is found in the 4th bit and $2^4=16$. (It should be noted that the 8 bits in a byte are numbered 0 through 7, from right to left.) Below is a sample program which uses REGISTER 1 and the binary switch setting.

```
100 WATCH1 1,16,16
110 WHEN #6 GOSUB 150
120 RTCS 1
130 PRINT "PROGRAM IS IN PROGRESS"
140 GOTO 130
150 PRINT "THE BINARY SWITCH IS IN THE
ENTER POSITION."
160 PRINT "THE PROGRAM WILL NOT RESUME
EXECUTION"
170 PRINT "UNTIL IT IS RETURNED TO THE
EXIT POSITION."
180 WAIT 1,16
190 RETURN
```

This program displays the message "PROGRAM IS IN PROGRESS" repeatedly. If the binary switch is put into the ENTER position, another message will be displayed and the program will not continue until the switch is put back into the EXIT position.

Line 100 sets up REGISTER 1, where the location is 1, the AND value is 16 (2^4 = 16) and the TEST value is 16. Assume location 1 contains the value of 63. This value is represented in binary code as 00111111. Notice that the fifth bit from the right (bit #4) contains a 1. Therefore, the binary switch is in the EXIT position. When line 100 is executed, Comm-Stor EXCLUSIVE OR's this value with the test value. This can be illustrated by:

```
      00111111
XOR   00010000
      00101111
```

The result of this operation yields the value 47. Next, Comm-Stor ANDs this result with the AND value, illustrated by:

```
      00101111
AND   00010000
      00000000
```

The result of this operation is equal to 0. Comm-Stor performs these operations every 16 milliseconds (if interrupts are enabled). If the final result is equal to zero, no interrupt will occur. Any non-zero result produces a "true" condition and an interrupt will occur. Line 110 tells Comm-Stor to execute the subroutine starting at line 150 if a REGISTER 1 interrupt occurs. Line 120 enables interrupts. Lines 130 and 140 produce an infinite loop. (The only way this program can end is if a [CNTRL/T] is issued.) Lines 150 - 170 display a message that the binary switch was put into the ENTER position. As this may cause problems in some applications, line 180 is used to delay execution of the program until the switch is put back into the EXIT position.

If this program is running and the binary switch is moved to the ENTER position, the fourth bit in location 1 is now

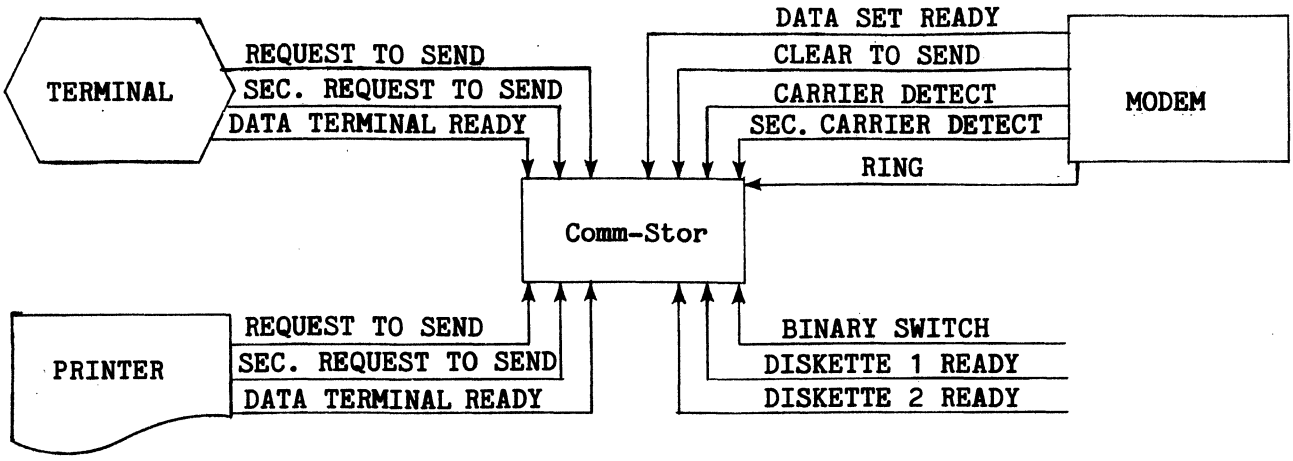

TERMINAL
REQUEST TO SEND
SEC. REQUEST TO SEND
DATA TERMINAL READY
DATA SET READY
CLEAR TO SEND
CARRIER DETECT
SEC. CARRIER DETECT
RING
MODEM
Comm-Stor
PRINTER
REQUEST TO SEND
SEC. REQUEST TO SEND
DATA TERMINAL READY
BINARY SWITCH
DISKETTE 1 READY
DISKETTE 2 READY

FIGURE #6-1

"turned off" (i.e. its value is 0). A sample decimal value containing a zero in this position might be 44,) which is 00101100 represented in binary mode. Comm-Stor will then perform:

```
      00101100
XOR   00010000
      00111100
```

and:

```
      00111100
AND   00010000
      00010000
```

This produces the result of 16 (which is a nonzero quantity and an interrupt will occur). Program control will then be passed to the subroutine starting at line 150. The message will be displayed and Comm-Stor will then execute the WAIT command. As mentioned on page 3-3, this command causes a BASIC program to delay execution until the specified location reaches an appropriate value. This is the major difference between the WAIT and WATCH commands. Therefore, if the binary switch is still in the ENTER position, line 180 causes Comm-Stor to perform:

```
      00101100
XOR   00000000
      00101100
```

and:

```
      00101100
AND   00010000
      00000000
```

This yields a value of zero and the program will delay. If the binary switch is returned to the EXIT position, location 1 will again contain the value of 63 and Comm-Stor will then perform:

```
      00111111
XOR   00000000
      00111111
```

and:

```
      00111111
AND   00010000
      00010000
```

This produces a nonzero value and line 190 will then be executed.

The RETURN command causes the program to resume execution from the line at which it left off when the REGISTER 1 interrupt occurred. Therefore, control will be transferred to either line 130 or 140, which causes the infinite loop to resume.

CHAPTER 7

COMMAND DEFINITIONS

All commands are described in this chapter and are accompanied by their respective formats where: <xxxx> = the description of the parameter, xxx = the actual (literal) parameter, [xxx] = the parameter is optional, and {xxx / yyy} = either xxx or yyy.

APPEND

Format: APPEND#<ref>

where: ref = A file reference number (4-24) associated with a disk file which may be a numeric constant, numeric variable, or numeric expression.

Purpose:

To start the processing of a disk file immediately after the last character in the file.

Description:

When the APPEND command is executed, the pointer associated with the disk file (related to the specified file reference number) is positioned immediately to the right of the last character in the file. This allows new data to be added to existing data in a file. See Chapter 5, The File Manager, for a more detailed description of this command.

Examples:

1.) APPEND #4 — Positions the pointer associated with the disk file (related to file reference #4) immediately to the right of the last character in the file.

2.) APPEND #I — Same as example 1, except the file reference number is the value of I.

3.) APPEND #(I+1) — Same as examples 1 and 2, except the file reference number is the value of I+1.

ASSIGN

Format:

ASSIGN #<ref>TO[(<drive>)] {*<trk>,<sec> / <file name>}

[,<char/rec>][,W][&<sec/buff>[,D]][;#...]

where:

ref = A file reference number (4-24) which may be a numeric constant or numeric variable.

drive = 1 for the upper drive or 2 for the lower drive; if this option is not specified, it defaults to 1.

file name = The name of a disk file which may be a character constant or a character variable.

trk = A track on the diskette which may be a numeric constant or numeric variable.

sec = A sector within the specified track on the diskette which may be a numeric constant or numeric variable.

char/rec = The number of characters per record which may be a numeric constant, numeric variable, or numeric expression; if this option is not specified, it defaults to 128.

W = If the disk file is "Write Protected", no error will occur when outputting to the file.

sec/buff = The number of sectors to be used with the file reference number which may be a numeric constant, numeric variable, or numeric expression; if this option is not specified, it defaults to 1 if File Buffer space has not been previously allocated for the file reference number.

D = Double buffered.

Purpose:

To assign a disk file or a track and sector to a file reference number.

Description:

The ASSIGN command may operate two different ways, depending on how the contents of the File Buffer have been allocated.

Case 1: Assume an OPEN command has only allocated space in the File Buffer for a file reference number, such as:

```
OPEN #6 TO+&4,D
```

(which reserves eight sectors, plus 35 bytes for a file header in the File Buffer) and the file reference number has been CLOSEd or RELEASEd. The File Buffer may be illustrated as follows:

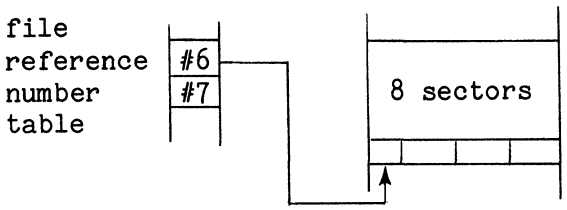


FIGURE #7-1

A subsequent ASSIGN command using the same file reference number will cause only certain parameters to be recorded into the corresponding file header. For example, if the following command is executed:

```
ASSIGN #6 TO "TEST",W
```

the File Buffer may now be illustrated as shown in Figure #7-2.

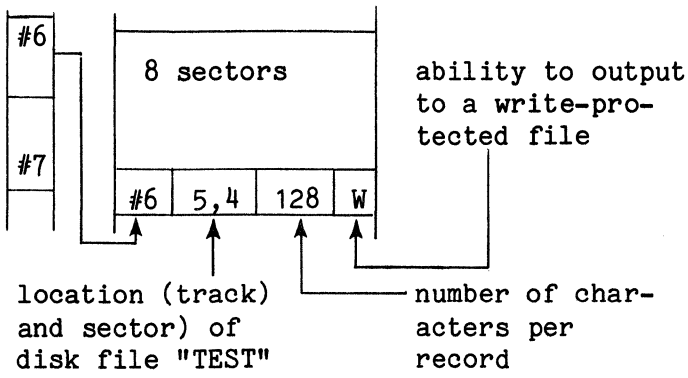


FIGURE #7-2

If the "sec/buff" option had been specified in the ASSIGN command, an error would have occurred because File Buffer space had previously been allocated for file reference number 6.

Case 2: Assume an OPEN command has allocated space in the File Buffer for a file reference number and has also related a disk file to the file reference number, such as:

```
OPEN #8 TO "DATA",40,W&7
```

the File Buffer may be illustrated as follows:

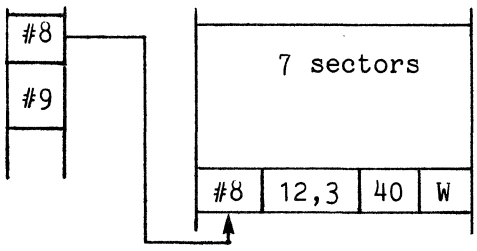


FIGURE #7-3

As can be concluded from the above diagram, a subsequent ASSIGN command would cause an error to occur because the file reference number already has disk file parameters associated with it. Therefore, a CLOSE or RELEASE command must be executed prior to an ASSIGN command.

If a CLOSE command is executed, the File Buffer can be illustrated as:

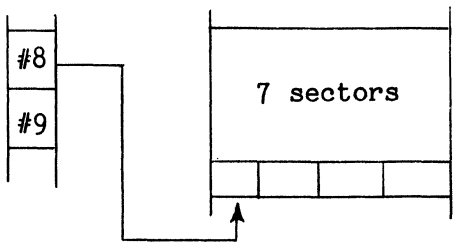


FIGURE #7-4

The file header has been cleared of its contents and the pointer from the file reference number table is still directed to the file header. This signifies that this File Buffer is still associated with file reference number 8, but that the file reference number has been CLOSEd (deactivating disk file "DATA".)

An ASSIGN command using file reference number 8, such as:

```
ASSIGN #8 TO "DATA2"
```

may now be executed, causing the File Buffer to appear as follows:

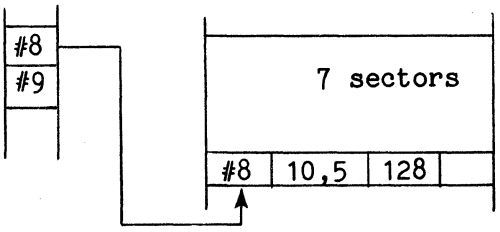


FIGURE #7-5

Now assume a RELEASE command is executed. The File Buffer now will appear as in Figure #7-6.

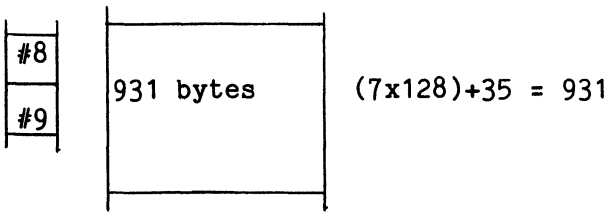


FIGURE #7-6

There are now 931 bytes <u>not</u> associated with any file reference number(s). A subsequent ASSIGN command will not only associate a portion of the diskette with the file reference number, but will also allocate space in the File Buffer for that reference number. For example,

```
ASSIGN #7 TO "DATA2",W
```

will cause the File Buffer to appear as shown below:

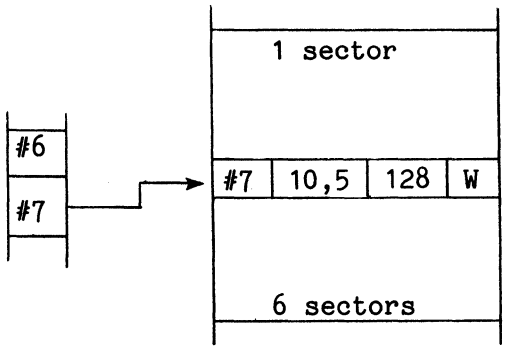


FIGURE #7-7

As there are six sectors of unused File Buffer space, another ASSIGN command, such as:

```
ASSIGN #6 TO "TEST",80&3
```

can be executed. The File Buffer will then appear like Figure # 7-8.

ASSIGN

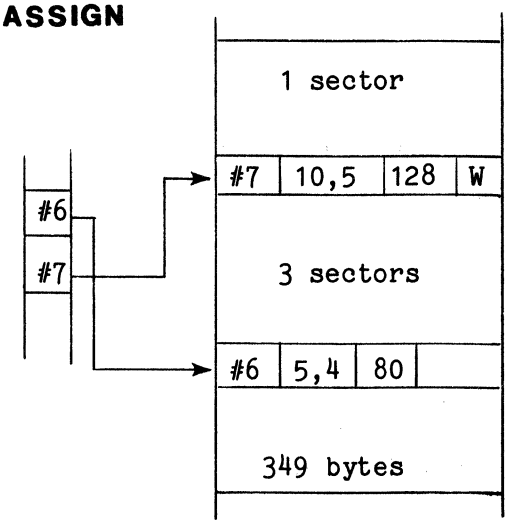


FIGURE #7-8

When a track and sector are specified in this command, everything will be performed as described above; however, a track and sector will be used instead of a disk file. ASSIGNing a track and sector to a file reference number gives the programmer the ability to access that track and sector through the end (Track 76, Sector 26) of the diskette. Please refer to Chapter 5 for a further description of this command.

Examples:

1.) **ASSIGN #4 TO "TEST"**

Associates disk file "TEST" with file reference number 4 and uses 1 sector plus a file header in the File Buffer.

2.) **ASSIGN #4 TO "TEST";#5 TO *1,3&4**

Same as example 1. It further associates the third sector of track 1 with file reference number 5 and uses four sectors plus a file header in the File Buffer.

3.) **ASSIGN #I TO (2) A$,50,W**

Associates the disk file on the lower drive, whose name is contained in A$, with the file reference number which is the value of I (specifying 50 characters per record and the ability to output to the disk file if it is Write Protected) and uses 1 sector plus a file header in the File Buffer.

4.) **ASSIGN #I TO (2) A$,50,W&5,D**

Same as example 4, except it uses 10 sectors (5 sectors, double buffered) plus a file header in the File Buffer.

AUTO

Format: **AUTO [<line num>][,<inc>]**

where: line num = the number of the first BASIC program line to be prompted; the default value is 100.

inc = the increment used for each successive line number; the default value is 10.

Purpose:

To automatically prompt line numbers when entering a BASIC program from the terminal.

Description:

Entering AUTO and optionally a line number and/or increment, will cause the line number specified to be displayed. A BASIC program line may then be entered. Entering a [CNTRL/X] (Line Delete character) will cause the current line to be deleted and its line number to be redisplayed. Upon entering the [RETURN] character, another line number will be displayed. This line number will be equal to the previous line number plus the specified increment. This will continue until the [RETURN] character is entered immediately after a supplied line number. Both the line number and the increment must be

positive integers when the AUTO command is issued.

AUTO

Examples:

1.) AUTO — Start with line 100, increment by 10.

2.) AUTO 40 — Start with line 40, increment by 10.

3.) AUTO 25,1 — Start with line 25, increment by 1.

4.) AUTO ,5 — Start with line 100, increment by 5.

BYE

Format: BYE

Purpose:

To exit the BASIC mode and return to Comm-Stor mode.

Description:

The BYE command will cause Comm-Stor to exit the BASIC mode and enter the Comm-Stor mode without clearing the entire User Memory Space. This means that:

1.) A BASIC program currently resident in memory will be left intact and may be RUN without reloading.

2.) Variable and File Space will not be cleared.

3.) Data currently residing in the Terminal or Modem Buffer will be cleared.

If TRACE is active when BYE is issued, it will not be automatically restored when the BASIC mode is re-entered (via .BA).

Example: BYE

CALL

Format: CALL [(<drive>)] <file name>

where: drive = 1 for the upper drive or 2 for the lower drive; if this option is not specified, it will default to 1.

file name = The name of a BASIC program saved in packed format which may be a character constant, character variable, or character expression.

Purpose:

To cause the loading and execution of a BASIC program, with the ability to restore and continue the BASIC program which requested the CALL.

Description:

This command, when issued from a BASIC program saved in packed format, causes Comm-Stor to LINK to the specified file (i.e. the file is loaded into the Program Text Buffer and executed without clearing the BASIC Work Space or the Fixed Space.) A RETURN command is used to return to the program that issued the CALL. That program will be restored (transferred from disk back into the Program Text Buffer) and execution will resume following the CALL command.

A CALLed program may, in turn, CALL another program. The number of CALLs that may be executed in this manner is limited only by the depth of the stack. Since only one program is resident in memory at one time, programs written to be CALLed (usually subroutines) may have line numbers that are the same as those in the program which issued the CALL without resulting in duplicate line numbers.

CALL

Variables are treated "globally". That is, variables created by a program will remain intact when another program is CALLed and may be freely used by the CALLed program. Similarly, variables created by the CALLed program will remain intact when the program RETURNs to the program which issued the CALL. After a CALL (and also after a RETURN) is executed, all user-defined functions and pending IF ERROR# commands are cleared.

It should be noted that if a CLEAR operation is performed within a CALLed program, an "ERROR - RG" will occur (because the stack is cleared) when the RETURN command is executed.

Examples:

1.) **CALL "SUB3"**

CALLs the file "SUB3".

2.) **CALL A$**

CALLs the file whose name is contained in A$.

3.) **CALL (2) LEFT$ (A$,5)**

CALLs the program located on the lower drive, whose file name is the first 5 characters in A$.

CLEAR

Format: **CLEAR [#<ref>]**

where: ref = a port reference number (0-3) as follows:

0 - invoker
1 - terminal
2 - modem
3 - printer

CLEAR

Purpose:

To clear BASIC Work Space, or the Terminal or Modem or Printer Buffer.

Description:

The CLEAR command without an argument will clear the contents of BASIC Work Space leaving the BASIC program, the contents of the Terminal, Modem, Printer and the Object Buffers intact. The CLEAR command may also be used to clear only the Terminal Buffer, or Modem Buffer, or Printer Buffer by specifying the appropriate reference number. The Invoker Buffer (Invoker is the port that issued the .BA command) may be cleared by specifying reference number 0.

Note: A CLEAR command is executed <u>internally</u> when:

1.) A BASIC line is added, deleted or edited,

2.) A RUN is executed,

3.) An OPEN #2 (Modem) is executed.

Examples:

1.) **CLEAR** Clear BASIC Work Space

2.) **CLEAR #1** Clear Terminal Buffer

3.) **CLEAR #I** Clear the buffer whose port reference number is the value of I.

Figure #7-9 reveals the changes that occur in User Memory after a CLEAR command is executed.

CLEAR

USER MEMORY SPACE

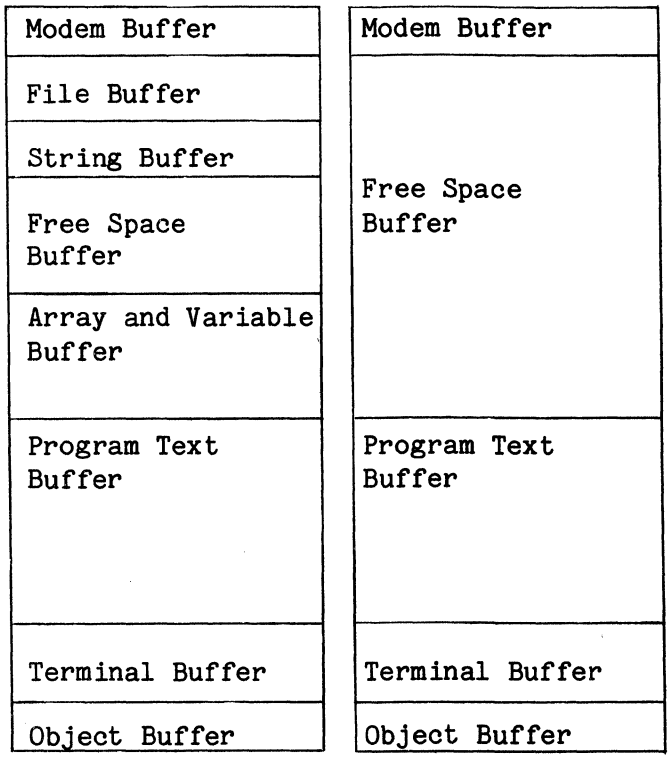


FIGURE #7-9

CLEARV

Format: CLEARV

Purpose:

To clear Variable Space.

Description:

The CLEARV command results in the clearing of all variables, arrays, and strings. This leaves the Modem, Terminal, Object, File, and Program Text Buffers intact. Figure #7-10 indicates the changes that occur in User Memory after a CLEARV command is executed.

CLOSE

Example: CLEARV

USER MEMORY SPACE

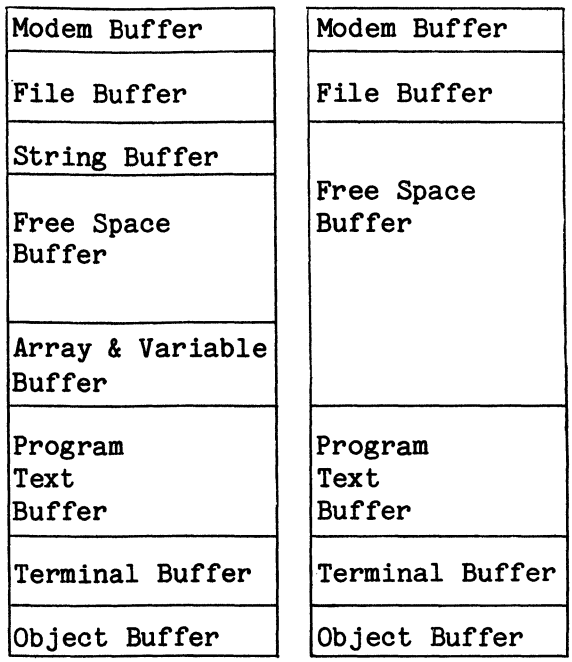


FIGURE #7-10

CLOSE

Format: CLOSE [#<ref>[,T][;#...]]

where: ref = A file reference number (4-24) associated with a disk file which may be a numeric constant or numeric variable.

T = Truncate

Purpose:

To deactivate a file previously used in an OPEN or ASSIGN command.

CLOSE

CLOSE

Description:

Issuing a CLOSE command causes Comm-Stor to deactivate a disk file (referenced by an OPEN or ASSIGN command) from a BASIC program. If a CLOSE alone is issued, all active files will be automatically deactivated. If a reference number is specified in the command, only the file associated with that reference number will be CLOSEd.

When a CLOSE is executed, the buffer size specified in the OPEN or ASSIGN command will still be associated with the file reference number(s). That same reference number may be ASSIGNed further on in the program with no specified buffer size (the buffer size will be the same as the originally OPENed size).

If a "T" is specified in the CLOSE command, any unused sectors within the file(s) will be released for further file space on the diskette. This will happen immediately if the file is the last directory entry on the disk. If it is not, the file will be truncated when a Comm-Stor SQUISH (.SQ) command is executed.

Examples:

1.) CLOSE — CLOSEs all files that are open.

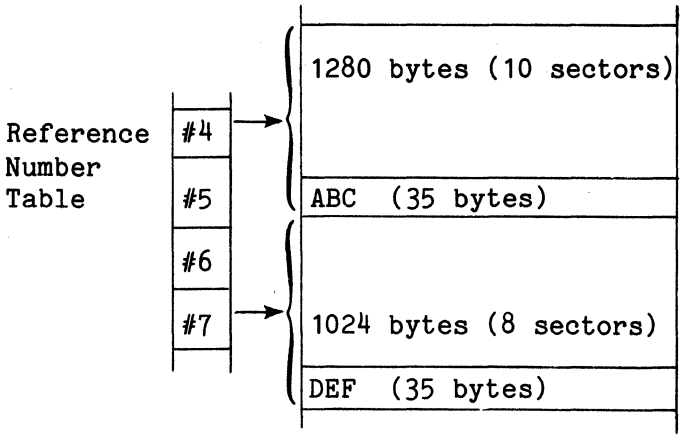
2.) CLOSE #4 — CLOSEs file reference number 4.

3.) CLOSE #4;#5;#I — CLOSEs file reference numbers 4,5, and the value of I.

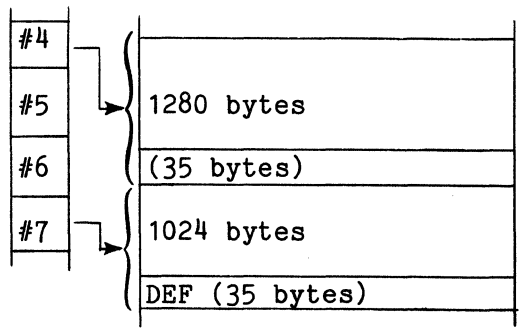
4.) CLOSE #4,T;#5 — CLOSEs file reference numbers 4 and 5 and truncates any unused sectors in the disk file which were associated with reference number 4.

EXAMPLE OF FILE BUFFER ALLOCATION

OPEN #4 TO "ABC"&10;#7 TO "DEF"&8



CLOSE #4



ASSIGN #4 TO "XYZ"

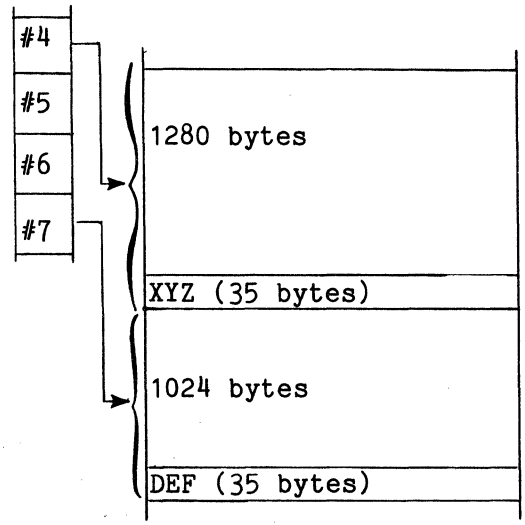


FIGURE #7-11

CLOSE

Sample Program:

```
10 CREATE "SAM",128
20 OPEN #4 TO "SAM"
30 PRINT #4;"THIS IS IT"
40 CLOSE #4
50 END
```

Line 10 CREATEs a disk file called "SAM" which is one sector long. The file "SAM" is related to file reference number 4 in line 20 and one sector plus 35 bytes are reserved in the File Buffer for reference number 4. Line 30 causes the characters "THIS IS IT" to be written to file "SAM" (through reference number 4). "SAM" is then CLOSEd (or deactivated) in line 40.

COM

Format:

COM [#<ref>[&#<ref>[&#..]];]<command>

where: ref = a number between 0 and 24 representing the destination (port or disk file) of any output data generated by the Comm-Stor command being issued.

Ref	Destination
0	Invoker (the port which issued the command to cause Comm-Stor to enter the BASIC mode). If the invoker is the Comm-Stor User Command Table, the output will be directed to the terminal.
1	Terminal
2	Modem
3	Printer
4-24	Diskette File (only one per command)

command = any valid Comm-Stor command, except .BA or .AL, which may be a character constant, character variable or character expression.

Purpose:

To execute Comm-Stor commands while in BASIC mode.

Description:

Any Comm-Stor command, except .BA or .AL, may be issued under BASIC control. Output generated as a result of the command may be directed to one or more ports, and/or to a diskette file. If more than one reference number is specified, only one may be assigned to a diskette file. For example, COM #4&#5 is illegal. (Note that it will not be flagged as an error, the right-most reference specified will be used, in this case, #5.)

Examples:

1.) COM".DS"

Issues the Comm-Stor command to Display Status.

2.) COM D$

Issues the Comm-Stor command which is contained in the string D$.

3.) COM #3;".DS"

Issues the Comm-Stor command to Display Status on the printer.

4.) COM #3&#4;".DS"

Issues the Comm-Stor command to Display Status on the printer and store it in the file associated with file reference number 4.

COM

5.) COM #0;".DD *"

Output entire directory to the port that commanded Comm-Stor to enter the BASIC mode (invoker).

CONT

Format: CONT

Purpose:

To continue execution of a program previously interrupted by a STOP command or a [CNTRL/T].

Description:

The CONT command, issued after a STOP command has been executed or after a [CNTRL/T] has been issued, will cause the program to resume execution starting at the line following the last line that was executed. The CONT command may not be used after:

1. a line is entered to modify the program,
2. a NEW command is executed,
3. a CLEAR command is executed,
4. a LOAD, MERGE, PRESS, or LINK command is executed,
5. an OPEN #2 is executed.

Entering a CONT command under any of the above conditions will cause an "ERROR - CN" to occur.

Example: CONT

CREATE

Format:

CREATE[(<drive>)]<file name>[,{M | <char/file>}]
[;[(<drive>)]...]

CREATE

where: drive = 1 for the upper drive or 2 for the lower drive; if the option is not specified, it defaults to 1.

file name = The name of a disk file which may be a string literal or variable.

char/file = The number of characters per file which will be rounded to the next multiple of 128 (sector). (i.e., 115 = 128 characters (1 sector), 258 = 384 characters (3 sectors).

M = The maximum space available.

Purpose:

To create a Directory entry (disk file) and allocate disk space to it.

Description:

On execution of the CREATE command, a directory entry is set up for the given file name with the specified amount of disk space being allocated to that file. If the "char/file" of "M" parameter is omitted, the amount of space allocated will be set to the "average file size" (described on page 1-5). Specifying "M" will result in allocating the largest continuous area of diskette space to that file.

The following is a list of potential error situations that may occur when a CREATE command is issued.

CREATE

Error	Description
BD (BAD DRIVE "NUMBER)	If "Drive #" is not 1,2 or a space
NO ROOM	Not enough room for the Directory entry's disk space
DE (DUPLICATE ENTRY)	A file with the specified name already exists

Examples:

1.) **CREATE "SAM"**

Create a Directory entry named SAM on drive 1. The "average file length" is the amount of disk space allocated to it.

2.) **CREATE (2) "BOBBY,500**

Create a Directory entry named BOBBY on drive 1 allocating 512 bytes (128*4) or four sectors of disk space to it.

3.) **CREATE "TEST",M**

Create a Directory entry named TEST on drive 1. The largest available contiguous diskette space is allocated to it.

4.) **CREATE (2)"DATA",100;A$,600**

Create a Directory entry named DATA on drive 2 allocating 128 bytes of disk space. Create a Directory entry named by the string A$ on drive 1 allocating 640 bytes of disk space.

DATA

Format: **DATA <val>[,<val>[,...]]**

where: val = A numeric or character constant.

DATA

Purpose: To prepare data for READ commands.

Description:

When the first READ command in a program is executed, an internal data pointer is associated with the first value in the first DATA command. (DATA commands may be placed anywhere in a program.) This first value is assigned to the variable specified in the READ command; the pointer then moves to the next value in the DATA command.

If all DATA commands in a program have been accessed and another READ command is executed, an "ERROR - OD" will occur unless a RESTORE command has been executed prior to the READ.

A comma terminates a data item, except when the comma is enclosed within quotation marks. Quotes are needed only for character data items when the character string contains commas, colons, or leading or trailing spaces.

If a numeric variable is specified in a READ command and the internal data pointer corresponds with a character constant, an "ERROR - TM" will occur in the line number of the DATA command.

Examples:

1.) **DATA 1** = Provides the value 1 for a READ command.

2.) **DATA A** = Provides the character "A" for a READ command.

3.) **DATA "C,2"** = Provides the characters "C,2" for a READ command.

4.) **DATA 1,A,"C,2"** = Provides the value 1, the character "A", and the characters "C,2" for a READ command.

DEF

Format: DEF FN<name>[(<arg>)]=<num exp>

where: name = Any numeric variable name joined to the characters "FN" to form the name of the function.

arg = The argument of the function which may be a numeric variable name.

num exp = A numeric expression.

Purpose:

To define a user-written numeric function.

Description:

The DEF command must be executed prior to any reference to the function name. The DEF command allows the programmer to define functional expressions similar to COS(X) or INT(X).

An argument is not required in the DEF command, but if one is specified, its value will be set equal to the value passed to the function. Additionally, the argument is a dummy variable. The argument may have the same name as a numeric variable used in the program, but the two are not considered to be the same variable. The numeric expression is one which contains the argument but may also contain program variables, arithmetic functions and other user-defined functions. It may not, however, contain a reference to its own function either directly or indirectly.

If an argument is not specified in the DEF command, all variables used in the user-defined function will be regular program variables.

A user-defined function may appear in any command containing a numeric expression.

Examples:

1.) **DEF FNA=X+1**

Defines a user function whose name is FNA which is equal to the program variable, X plus one.

2.) **DEF FNB3(Y)=Y/2**

Defines a user function, whose name is FNB3 which is equal to the value passed to the function divided by two.

3.) **DEF FNX=FNB3 (A%)+X**

Defines a user function, whose name is FNX which is equal to X plus the result of FNB3 (A%).

DEL

Format: DEL<range>

where: range = A range of line numbers as described in the LIST command.

Purpose:

To delete a range of lines from a BASIC program.

Description:

When this command is executed, only those BASIC program lines whose line numbers are included in the range will be deleted from the Program Text Buffer. The following conditions apply:

1. Specifying only one line number will delete only that line number and the corresponding line from the program.

2. If only one line number is specified and a hyphen precedes it, only those line numbers (and their corresponding lines) greater than the one specified will remain in memory.

3. If only one line number is specified and a hyphen follows it, only those line numbers (and their corresponding lines) less than the one specified will remain in memory.

4. If two line numbers are specified (separated by a hyphen), the first one must be less than or equal to the second one and all line numbers (and their corresponding lines) greater than or equal to the first one and less than or equal to the second one will be deleted from memory.

5. If no line numbers in the program are included in the specified range, Comm-Stor will return to an idle state.

Examples:

1.) **DEL 20** Deletes line 10 from the current program.

2.) **DEL 10-50** Deletes line 10 through 50 in the current program.

3.) **DEL - 100** Deletes the first line through line 100 of the current program.

DIM

Format: DIM<array name>(<rows>[,<columns>

[,...]])[,<array name>...]

where: array name = Any valid variable name.

rows = The number of rows in the array.

columns = The number of columns in the array.

Purpose:

To allocate space in memory for an array.

Description:

When this command is executed, memory space is allocated in the Array and Variable Buffer for the entire array and each element is initialized to zero (or null strings for character arrays). A total of 255 dimensions are allowed for an array, providing each dimension is not greater than 32767 and there is sufficient User Memory Space to accommodate the array.

After an array has been dimensioned, it cannot be redimensioned or an "ERROR - DD" will occur unless an ERASE command (or a CLEAR or CLEARV operation) is executed.

Examples:

1.) DIM A$(2)

Dimensions the character array, A$, to have three elements: A$(0), A$(1), and A$(2).

2.) DIM A3(K),D$(20,40,60)

Dimensions the numeric array, A3, to have K+1 elements and the character array, D$, to have elements: D$(0,0,0),D$(1,0,0),...,D$(19,40,60), and D$(20,40,60).

3.) DIM A3(B(N,J),I-2)

Dimensions numeric array A3 to have (I-2)+1 columns and the number of rows equals the value of the array element (B(N,J))+1.

DIM#

Format:

DIM #<ref> TO { <num array> / <char array> }

{ (<rows>[,<cols>[,...]]) / (<rows>[,<cols>[,...]]) [=<num exp>] }

[,...][;#...]

where: ref = A file reference number (4-24) associated with a disk file and the specified array(s) which may be a numeric constant, numeric variable or numeric expression.

num array = A numeric variable name.

char array = A character variable name.

rows = The number of rows in the array.

cols = The number of columns in the array.

num exp = A numeric expression.

Purpose:

To associate a virtual array with a file reference number.

Description:

When this command is executed, the size of the virtual array(s) is stored in memory, but the contents of the array(s) are not. The values for the array(s) are obtained when the file reference number is associated with a disk file. A reference to an element of an array causes Comm-Stor to obtain the corresponding value from the disk file. When an element of a virtual array is changed, Comm-Stor automatically changes the corresponding value in the disk file.

If a character array is specified in this command, Comm-Stor will automatically associate 32 characters in the disk file with each element of the array. If this DIM# is not desired, the programmer can specify the length of each element. For example,

```
DIM#4 TO A$(25)=50
```

will cause Comm-Stor to associate 50 characters in the disk file associated with file reference number 4 with each element of virtual array A$.

The virtual array(s) will be associated with the specified file reference number unless an ERASE command (or a CLEAR or a CLEARV operation) is executed.

Refer to Chapter 5 for a more detailed description of virtual arrays.

Examples:

1.) `DIM #6 TO A3(6)`

Dimensions the numeric virtual array A3 (to have 7 elements) and obtains the values for these elements from the disk file associated with reference number 6.

2.) `DIM #I TO A3(A(N))`

Dimensions the numeric virtual array A3 (to have the value of the array element A(N) plus 1 elements) and obtains the values for these elements from the disk file associated with the reference number whose value is I.

3.) `DIM #I TO A3(A(N));D$(9)`

Same as example #2, but also dimensions character array D$ to have 10 elements.

4.) `DIM #ITO A3(A(N));D$(9);#9TO A$(2),B3(7)`

Same as example #3, but also dimensions virtual arrays A$ and B3 whose elements are obtained from the disk file associated with reference number 9.

EDIT

Format: EDIT [<range>]

EDIT

EDIT

where: range = A range of line numbers as described in the LIST command.

Purpose:

To alter BASIC program lines which reside in the Program Text Buffer.

Description:

Upon issuing the EDIT command, the first line that is included in the range will be displayed. At this time, any options listed in Table #7-1 may be entered from the terminal device.

If a character was entered but does not exist in the line, the line will remain unchanged. The EDIT for the next line in the range will then be started or, if there are no more lines in the range, Comm-Stor will return to an idle state. If the character does exist, the left-most portion of the line (up to and including the character) will be displayed and any options listed in Table #7-2 may be entered.

Examples:

1.) **EDIT**

Starts the edit of all lines in the Program Text Buffer.

2.) **EDIT 10-**

Starts the edit for all lines (whose line numbers are equal to or greater than 10) in the Program Text Buffer.

3.) **EDIT-100**

Starts the edit for the lines in the Program Text Buffer, beginning with the first line through line 100.

4.) **EDIT 100-150**

Starts the edit for lines 100 through 150, residing in the Program Text Buffer.

Operation	Description
[RETURN]	causes the current line to remain unchanged and will start the EDIT for the next line.
[CNTRL/T]	causes the line to remain unchanged and will return Comm-Stor to an idle state.
any character	causes Comm-Stor to search for the first occurrence of that character.

TABLE #7-1

Operation	Description
[RETURN]	causes the remainder of the current line to be deleted, completing the EDIT for that line.
[CNTRL/I]	causes Comm-Stor to search for the next occurrence of that character. If another occurrence does not exist, the EDIT for that line will be complete. If the character does exist, any options in this table may again be entered.
[CNTRL/Y]	allows the SEARCH character to be changed to the next character typed after a [CNTRL/Y] is entered.
[ESC]	causes the next character in the line to be displayed and any option in this table may again be entered. If another character does not exist in the line, the EDIT for that line will be complete.

Operation	Description
[CNTRL/T]	causes the line to remain unchanged and returns Comm-Stor to an idle state.
[DEL]	causes the last displayed character to be deleted from the line and any options listed in this table may now be entered.
[CNTRL/X]	causes the EDIT for that line to be started again, at which time any options listed in Table #7-1 may be entered.
any character(s)	causes the character(s) to be inserted to the right of the last character displayed and any options listed in this table may now be entered.

TABLE #7-2

END

Format: END

Purpose:

To terminate the execution of a BASIC program.

Description:

When an END command is executed, the program currently being executed is terminated. The program and variables will still reside in the Program Text Buffer and Variable Space and may be modified and/or re-executed. Additionally, any active files are automatically closed by Comm-Stor; the system will then return to the Direct Mode.

An END command is optional. Normally, when the highest numbered BASIC program line is executed, the program will automatically terminate in the manner described above.

Example: 550 END

ERASE

Format: ERASE<array list>

where: array list = A list of array names separated by commas.

Purpose:

To delete an array from the Array and Variable Buffer.

Description:

Upon execution of the ERASE command, each specified array will no longer exist. This allows the programmer to use the available room in memory for creating additional arrays or for creating a larger Free Space Buffer size.

Example:

1.) ERASE A3 Erases array A3.

2.) ERASE A3,A$,BB Erases arrays A3, A$, and BB.

FOR...TO

Format:

FOR<control var>=<init val>TO<final val>

[STEP<inc>]

where: control var = The control variable for the loop; it may be any numeric variable.

init val = A numeric constant, numeric variable, or numeric expression whose value initializes the control variable.

final var = A numeric constant, numeric variable, or numeric expression whose value terminates the loop when exceeded by the value of the control variable.

inc = A numeric constant, numeric variable, or numeric expression whose value is the increment (positive or negative) for the control variable when a NEXT command is executed; defaults to 1 if this option is omitted.

Purpose:

To allow repetition of a sequence of BASIC program lines.

FOR...TO

FOR...TO

Description:

When this command is executed, the control variable is set equal to the initial value. It is then checked to see if it has exceeded the value of the final value. If it has, program control will be transferred to the line following the corresponding NEXT command. If the control variable has not exceeded the value of the final value, the BASIC lines following the FOR...TO command will be executed until the corresponding NEXT command is encountered. The NEXT command causes the control variable to be incremented by the value of the specified increment. The control variable is then checked against the final value. If it has exceeded the final value, program control will be transferred to the BASIC line following the NEXT command. If it has not exceeded the final value, control will then be transferred to the line following the FOR...TO command.

FOR/NEXT loops can be nested. That is, a loop can contain another loop. Nesting may occur up to the depth of the stack (approximately 14 deep). The following is an example of three-deep nesting:

```
50 FOR I=1 TO 10 STEP 2
 .
 .
90 FOR J=1 TO 6
 .
 .
130 FOR K=A TO B STEP A/B
 .
 .
170 NEXT K
 .
 .
200 NEXT J
 .
 .
250 NEXT I
```

The following loops are illegal:

```
10 FOR I=1 TO 5
 .
 .
50 FOR J=A TO B
 .
 .
100 NEXT I
110 NEXT J
```

If a control variable is not specified in a NEXT command, it is assumed to be the control variable of the last executed FOR...TO command. If a control variable is specified, the system will search for the corresponding FOR...TO command and execute that FOR/NEXT loop.

The value of the control variable may be changed within a loop. All other parameters, however, are set after the FOR...TO command is executed and cannot be modified.

Control may be transferred out of a loop. In such a case, care should be taken when returning to the loop that the FOR...TO command is not re-executed.

Examples:

1.) FOR I=1 TO 5

Specifies a loop that is dependent on I, whose initial value is 1 and whose final value is 5, using 1 as the increment value.

2.) FOR I=1 TO 5 STEP 2.5

Same as the above example, except 2.5 is the increment value.

3.) FOR I=A TO B STEP C

Specifies a loop to be dependent on I, whose initial value is the value of A and whose final value is the value of B, using the value of C as the increment value.

FOR...TO

4.) FOR I=A(N) TO C/5 STEP H+1

Specifies a loop to be dependent on I, whose initial value is the value of the array element A(N) and whose final value is the value of C divided by 5, using the value of H +1 as the increment value.

5.) FOR I=5 TO 4

As the final value is less than the initial value, control will be transferred to the statement after the corresponding NEXT statement.

GOSUB

Format: GOSUB<line num>

where: line num = A line number in a BASIC program.

Purpose:

To transfer program control to a subroutine starting at the specified line number.

Description:

When a GOSUB command is executed, control is transferred to the subroutine starting at the specified line number. (The subroutine may contain more than one RETURN command. However, when a RETURN is executed, control is transferred back to the line following the GOSUB command.) Nesting is permitted (i.e. a subroutine may be called from another subroutine). After each GOSUB is executed, the line number to which program control is to be returned is placed on the stack. When RETURN commands are encountered, the last line number placed on the stack is the first one to which control is transferred and cleared from the stack. If a RETURN command is executed and the stack is empty, an "ERROR - RG" will occur.

IF END #...

If the specified line number does not exist in the program, an "ERROR - US" will occur.

Examples:

1.) GOSUB 500 Transfers program control to the subroutine starting at line number 500.

GOTO

Format: GOTO<line num>

where: line num = A line number in a BASIC program.

Purpose:

To transfer program control to a specific line in a program.

Description:

When this command is executed, program control is transferred to the line corresponding to the specified line number. If the line number does not exist in the program, an "ERROR - US" will occur.

Examples:

1.) GOTO 500 Transfers program control to line number 500.

IF END #...

Format: IF END#<ref>GOTO { <line num> / END }

where: ref = A file reference number (4-24); may be a numeric constant or numeric variable.

line num = A line number in a BASIC program, which may be a numeric constant or a numeric variable.

IF END #...

END = Terminates the program detection of the end of file condition.

Purpose:

To transfer program control to a specified line number when an end of file condition occurs.

Description:

This command must be executed prior to the detection of the end of file condition. If the end of a disk file associated with the specified file reference number is reached, program control will be transferred to the specified line number. If END is specified in this command, the program detection of the end of file condition will be terminated.

Examples:

1.) IF END #4 GOTO 500

If the end of the disk file associated with file reference number 4 is reached, program control will be transferred to line number 500.

2.) IF END #I GOTO 60

If the end of the disk file associated with the file reference number, which is the value of I, is reached, control will be transferred to line number 60.

3.) IF END #4 GOTO J

If the end of the disk file associated with the file reference number 4 is reached, program control will be contained in line number J.

IF ERROR #...

Format: IF ERROR#<code>GOTO {<line num> | END}

where: code = The number related to an error condition (see Appendix G) which may be a numeric constant or numeric variable.

line num = A line number in a BASIC program which may be a numeric constant or a numeric variable.

END = Terminates the program detection of the specified error condition.

Purpose:

To transfer program control to a specific line number when a particular error condition is detected.

Description:

This command must be executed prior to the detection of the specified error condition. When this command is used, program control will be transferred to the specified line number when the error condition is detected. This will happen any time the error occurs, unless END is specified in another IF ERROR # command. Specifying END instead of a line number terminates program detection and control of the error condition.

Examples:

1.) IF ERROR #8 GOTO 600

If an "ERROR - NO FIND" occurs, program control will be transferred to line number 600.

2.) IF ERROR #8 GOTO END

Terminates program detection and control of an "ERROR - NO FIND".

3.) IF ERROR #I GOTO T

If an error occurs whose "code" is the value of I, program control will be transferred to the line number whose value is T.

IF...GOTO

Format: IF <rel exp>GOTO<line num>

where: rel exp = A relational expression.

line num = A line number in a BASIC program.

Purpose:

To transfer program control based on the evaluation of a relational expression.

Description:

Upon execution of the IF...GOTO command, the relational expression is evaluated. If the result is true, program control will be transferred to the specified line number. If the result is false, control is passed to the next line following the IF...GOTO command. If the specified line number does not exist in the program, an "ERROR - US" will occur.

Examples:

1.) IF A=B GOTO 500

If the value of A and B are equal, program control will be transferred to line number 500.

2.) IF LEFT$(A$,1)="*" GOTO 100

If an asterisk is the left-most character in A$, program control will be transferred to line number 100.

3.) IF A>K+2 GOTO 10

If the value of A is greater than the value of K plus 2, program control will be transferred to line number 10.

4.) IF NOT A = NOT (B OR C) GOTO 453

If the value of NOT A is equal to the value of NOT (B OR C), program control will be transferred to line number 453.

IF...THEN

Format:

IF<rel exp> THEN { <line num> / <command> }

[:<command>[:...]]

where: rel exp = A relational expression.

line num = A line number in a BASIC program.

command = A BASIC command.

Purpose:

To make a decision regarding the execution of BASIC commands based on the evaluation of a relational expression.

Description:

Upon execution of this command, the relational expression is evaluated. If the result is true, the specified BASIC commands will be executed or, if a line number is specified, program control will be transferred to that line number. If the result is false, control is passed to the line following the IF...THEN command.

Examples:

1.) IF A=B THEN PRINT A

IF...THEN

If the values of A and B are equal, the value of A will be displayed.

2.) **IF A>C THEN 500**

If the value of A is greater than the value of C, program control will be transferred to line number 500.

3.) **IF A$="ABC" THEN B$=A$: GOTO 500**

If A$ contains the characters "ABC", B$ will be assigned those characters and program control will be transferred to line number 500.

4.) **IF X=Z THEN Y=5:B=X:X=FNA(Z+3)**

If the value of X is equal to the value of Z, Y will be assigned the value of 5, B will be assigned the value of X, and X will be assigned the value of FNA(Z+3).

IF...THEN...ELSE

Format: **IF<rel exp>THEN** { **<line num>** / **<command>** }

[:<command>[:...]] ELSE<command>[:...]

where: rel exp = A relational expression.

line num = A line number in a BASIC program.

command = A BASIC command.

Purpose:

To make a decision regarding the execution of BASIC commands based on the evaluation of a relational expression.

Description:

The IF...THEN...ELSE command is similar to the IF...THEN command, except that if the relational expression is evaluated to be false, all of the commands following the "ELSE" will be executed.

IMAGE

Nesting IF...THEN...ELSE commands is acceptable, however, the command, including spaces, must not be longer than 255 characters. The programmer may have an IF...THEN command within an IF...THEN... ELSE command by specifying "ELSE ELSE".

Examples:

1.) **IF A=B THEN 400 ELSE PRINT A,B**

If the values of A and B are equal, program control will be transferred to line number 400. Otherwise, the values of A and B will be displayed.

2.) **IF A$=C$ THEN IF C=D THEN A=A+1 ELSE ELSE D$=A$**

If the contents of A$ and C$ are the same and if the values of C and D are equal, then A will be assigned the value of A+1. If the contents of A$ and C$ are not the same, D$ will be assigned the contents of A$.

3.) **IF X^3<>Z/Y THEN Z=Z-1:Y=Y+2 ELSE Z=Z+2:Y=Y-1**

If the value of X raised to the power of 3 does not equal the value of Z divided by the value of Y, Z will be decremented by 1 and Y will be incremented by 2. Otherwise, Z will be incremented by 2 and Y will be decremented by 1.

4.) **IF LEFT$(A$,1)="A" THEN B$=A$ ELSE B$=C$:GOTO 500**

If the first character in A$ is the character "A", then B$ will be assigned the characters in A$. Otherwise, B$ will be assigned the characters in C$ and program control will be transferred to line number 500.

IMAGE

Format: **IMAGE<format>**

IMAGE

where: format = The format used to input or output data.

Purpose:

To provide a format for inputting or outputting data.

Description:

This command provides a format to be used with PRINT USING, PRINT # USING, and READ # USING commands. The options available for use in an image are listed below:

#	- field position
n#	- repeating field position
S	- space fill-in (Print Using)
nS	- repeating space fill-in
S	- skip feature (Read Using)
nS	- repeat skip feature
N	- null
nN	- repeating null fill-in
()	- repeat feature
+,-	- sign control
$	- floating dollar sign control
n$	- repeating dollar sign
.	- decimal point control
B	- binary (transparent) format
P	- reposition function
T	- truncate field character
E	- exponential notation
I	- insertion character
D	- delimiter assignment
A	- delimiter allocation
X	- suppress echo at terminal

Please refer to Chapter 4 for a more detailed description of an IMAGE.

Example:

1.) 50 IMAGE 2S3#
Provides a format equal to two spaces and three character positions.

INPUT

Format: INPUT[<char constant>;]<var list>

where: char constant = A character constant.

var list = A list of variables, separated by commas.

Purpose:

To assign values entered from the terminal device to variables.

Description:

If a character constant is specified in this command, its contents will be displayed on the terminal. A question mark is displayed when the INPUT command is executed.

The values entered from the terminal must be separated by commas or a [RETURN] character. If less values are entered from the terminal than the number of variables in the variable list, two question marks, "??", will be displayed; this will occur until all of the variables are assigned values. If more values are entered than are requested, a warning "?EXTRA IGNORED" will be displayed and program execution will resume normally.

Character data which contains commas or leading or trailing spaces should be enclosed in quotes.

Examples:

1.) INPUT A

The value entered from the terminal will be assigned to A.

2.) INPUT "ARE YOU DONE";A$

The characters "ARE YOU DONE?" are displayed on the terminal and the characters entered from the terminal are assigned to A$.

KILL

Format:

KILL[(<drive>)]<file name>[;[(<drive>)] ...]

where: drive = 1 for the upper drive or 2 for the lower drive; if this option is not specified, it defaults to 1.

file name = The name of a disk file; may be a character constant or character variable.

Purpose:

To remove a Directory entry (disk file) and release file space on a diskette.

Description:

When the KILL command is executed, Comm-Stor searches for the specified disk file on the diskette inserted into the specified drive. If the file is not found, an "ERROR - NO FIND" will occur. If the file is found, it will be removed from the diskette in the same manner as the Comm-Stor CANCEL (.CN) command.

Examples:

1.) KILL "TEST"

Removes disk file "TEST" from the diskette inserted in the drive (upper drive of a dual drive unit).

2.) KILL "TEST";(2)A$

Same as example #1, but also removes the disk file on the lower drive (of a dual drive unit) whose name is contained in A$.

3.) KILL LEFT$(A$,7)

Removes the disk file on the drive (upper drive of a dual drive unit) whose name is the first 7 characters in A$.

LET

Format: [LET]<var>= { <var> / <constant> / <exp> }

where: var = A numeric variable or character variable.

constant = A numeric constant or a character constant.

exp = A numeric expression or character expression.

Purpose:

To assign a value to a variable.

Description:

When the LET command is executed (the characters "LET" are optional), Comm-Stor evaluates the variable, constant, or expression (to the right of the equal sign) and obtains a value. This value may be either a numeric value or a character string.

Comm-Stor then checks to see if the variable to the left of the equal sign, the ASSIGNMENT variable, is the same type as the value. If it is not, such as a numeric assignment variable and a character string value, an "ERROR - TM" will occur. The assignment variable may either be a variable or an element of an array. If the assignment variable is a numeric and the precision of it is not the same as the precision of the numeric value, the numeric value will be converted to the precision of the assignment variable. For example, if the assignment variable is an

integer and the numeric value is single precision, the numeric value will be changed into integer format when it is assigned to the variable.

Examples:

1.) LET A%=1*1.5

Determines the value of 1 multiplied by 1.5, which is 1.5, converts it to an integer value, 1, and assigns 1 to A%.

2.) A$=LEFT$("TEST",1)

Assigns the character "T" to A$.

3.) LET A=FNXYZ(Z)/2

Assigns the value of FNXYZ(Z) divided by 2 to numeric variable A.

4.) Z$=Z$+"123"

Joins the characters in Z$ to the characters "123" and assigns the result to character variable Z$.

LINK

Format: LINK[(<drive>)] <file name>

where: drive = 1 for the upper drive or 2 for the lower drive; it defaults to 1 if this option is not specified.

file name = The name of a disk file which contains a packed BASIC program; it may be a character constant or character variable.

Purpose:

To enter a program into the Program Text Buffer without affecting any other buffer except the Free Space Buffer, and execute it.

Description:

When the LINK command is executed, Comm-Stor searches for the BASIC program (saved under the specified file name) which is in packed format. It then determines if this program can fit into the User Memory Space. To do this, the current program's byte count plus the number of bytes in the Free Space Buffer is compared to the LINKed program's sector byte count. If the former is less than the latter, an "ERROR - OM" will occur. If the former is greater than or equal to the latter, the current program will be deleted from the Program Text Buffer and the LINKed program will be entered into that buffer. Any bytes needed to accommodate the LINKed program or any which are left over after the program is entered into the buffer will be obtained from or directed to the Free Space Buffer.

After the LINK is executed, any user-defined functions and program control of error conditions are cleared from the User Memory Space. The LINKed program will be automatically executed after being loaded into memory.

One LINKed program may LINK to another program with no limit to the number of LINKs.

Examples:

1.) LINK "PROG 2"

LINKs "PROG 2".

2.) LINK (2)A$

LINKs the file, whose name is contained in A$, on the lower drive of a dual drive unit.

LIST

Format:

LIST[#<ref>[&#<ref>[&#...]];][<range>]

where: ref = A port reference number (1-3), a file reference number (4-24), or 0 (for the invoker of the .BA command), any of which may be a numeric constant or a numeric variable.

range = One line number alone or one line number preceded or followed by a hyphen, or two line numbers separated by a hyphen.

Purpose:

To sequentially output a BASIC program's lines.

Description:

Upon execution of this command, the current program's lines will be output. If no reference numbers are specified, the output will be displayed on the terminal device. If reference numbers are specified, the output will be directed to the specified device(s) and/or disk file. Only one file reference number should appear in this command. If more than one is specified, the output will only be directed to the disk file associated with the last file reference number specified.

The following rules apply when specifying line numbers:

1.) If a range of line numbers is specified, only those line numbers included in that range will be output.

2.) If one line number alone is specified, only that line number (and the line associated with it) will be output.

3.) If only one line number is specified and a hyphen precedes it, the output will consist of all lines from the beginning of the program up to and including the specified line number (and corresponding line).

4.) If only one line number is specified and a hyphen follows it, the output will consist of that line number (and its corresponding line) and all other successive lines through the end of the program.

5.) If two line numbers are specified, only those line numbers included in the range will be output. If a range is not specified in this command, all lines of the current program will be output.

6.) If a line number cannot be found that is included in the range, the execution of this command will be terminated.

Examples:

1.) LIST

Displays the entire program on the terminal.

2.) LIST#1&#4;10-100

Outputs all line numbers from 10 through 100 to the terminal and to the disk file associated with file reference number 4.

3.) LIST #0;100-

Outputs lines 100 through the end of the current program on the device which the .BA command was issued.

4.) LIST #0&#I;-500

Outputs the lines from the beginning of the current program through line number 500 to the invoker's device and the disk file (or device) associated with the reference number whose value is I.

LOAD

Format:

LOAD[(<drive>)]<file name>[,R][,<range>]

where: drive = 1 for the upper drive or 2 for the lower drive; it defaults to 1 if this option is not specified.

file name = The name of a disk file which contains a BASIC program; it may be a character constant or a character variable.

R = RUN

range = A range of line numbers as described in the LIST command.

Purpose:

To enter a BASIC program into the Program Text Buffer.

Description:

When the LOAD command is issued in the Direct Mode, Comm-Stor searches the diskette inserted in the specified drive for the specified file name. (If no drive is specified, the upper drive is assumed.)

The following conditions apply to the LOAD command:

1.) The contents of the file are checked to see if they consist of BASIC program lines. If they do not, Comm-Stor will return to an idle state (or an "ERROR - PT" will occur if the file is not in packed format, text format, ASCII format, or in Sykes "RUN" format).

2.) If the contents do consist of BASIC program lines, the lines will be entered into the Program Text Buffer and the contents of the BASIC Work Space will be deleted (these bytes will be returned to the Free Space Buffer).

3.) If an "R" is specified in the LOAD command, the program will be automatically executed.

4.) When a range of line numbers is specified in the LOAD command, only the BASIC program lines included in the range will be entered into the Program Text Buffer provided the file is in text or ASCII format.

5.) If the file is in packed or Sykes "RUN" format, the specified range will be ignored.

6.) When the LOAD command is issued in the RUN Mode and an "R" is not specified, everything is done exactly as described above.

7.) If an "R" is specified, only the contents of the Variable Space will be deleted (these bytes will be returned to the Free Space Buffer); the File Buffer will not be affected in anyway.

Examples:

1.) **LOAD "PROG 1"**

The program "PROG 1" is loaded into the Program Text Buffer.

2.) **LOAD (2) A$,R**

The program (on the lower drive) whose name is contained in A$ is loaded into the Program Text Buffer and executed.

3.) **LOAD "PROG 1",R,10-68**

Lines 10 through 68 in the program "PROG 1" are loaded into the Program Text Buffer and executed.

MERGE

Format:

MERGE[(<drive>)]<file name>[,<range>]
[;[(<drive>)]...]

where: drive = 1 for the upper drive or 2 for the lower drive; it defaults to 1 if this option is not specified.

file name = The name of a disk file, which contains a BASIC program in text format; it may be a character constant or a character variable.

range = A range of line numbers as described in the LIST command.

Purpose:

To load a BASIC program into the Program Text Buffer without deleting the previous contents of that buffer.

Description:

When the MERGE command is issued, Comm-Stor searches the diskette inserted in the specified drive for the specified file name. (If no drive is specified, the upper drive is assumed.)

The following conditions apply to the MERGE command:

1.) If the file is in text format, the contents of the file will be entered into the Program Text Buffer. Any previous lines in the Program Text Buffer will remain in that buffer. If the file is not in text format, an "ERROR - PT" will occur.

2.) If a range of line numbers is specified in the MERGE command, only those lines included in the range will be entered into the Program Text Buffer.

3.) The line numbers of the merged file must be greater than those currently in the Program Text Buffer.

4.) When a MERGE is executed, a CLEAR operation is automatically executed by Comm-Stor. Therefore, it is advisable to put any BASIC program lines containing MERGE commands in the very beginning of the BASIC programs.

5.) If the MERGE command is issued in the RUN Mode, the MERGE (and implicit CLEAR) will be executed and the execution of the program will resume at the line following the MERGE command.

Examples:

1.) **MERGE "ABC"**

Adds the contents of the file "ABC" to the Program Text Buffer and performs a CLEAR operation.

2.) **MERGE (2)A$,10-100**

Adds lines 10 through 100 in the file (on the lower drive) whose name is contained in A$ to the contents of the Program Text Buffer and performs a CLEAR operation.

3.) **MERGE "ABC";(2)A$,10-100**

Performs a combination of the operations as described in the two above examples.

4.) **MERGE A$;B$;C$;D$;E$**

Merges the text programs whose file names are contained in A$,B$,C$,D$, and E$ into the Program Text Buffer and performs a CLEAR operation.

NEW

Format: **NEW[[(<drive>)]<file name>]**

NEW

where: drive = 1 for the upper drive or 2 for the lower drive; if not specified, this option defaults to 1.

file name = The name of the disk file, which holds a program, not enclosed in quotes.

Purpose:

To load and execute a program or to delete everything in the BASIC Program Space and return all bytes to the Free Space Buffer.

Description:

When this command is executed without an argument, everything in the BASIC Program Space is cleared.

If a file name is specified, the command is equivalent to:

LOAD "file name",R in Direct Mode.

The character @ is symbolically equivalent to the characters NEW and may be directly substituted (see example below).

Examples:

1.)	**NEW**	Clear BASIC Program Space
2.)	**NEW PROG 1**	Clear BASIC Program Space; load and execute the program "PROG 1"
3.)	**@PROG 1**	Same as example #2

NEXT

Format: NEXT [<control var>]

where: control var = The control variable corresponding to the FOR...TO command.

Purpose:

To end the loop of BASIC lines initiated by a FOR...TO command.

Description:

See the FOR...TO command.

ON...GOSUB

Format: ON<num exp> GOSUB<line num list>

where: num exp = A numeric constant, numeric variable, or numeric expression.

line num list = A list of line numbers in a program separated by commas.

Purpose:

To transfer program control to one of a group of subroutines based on the evaluation of a numeric expression.

Description:

When this command is executed, the numeric constant, numeric variable, or numeric expression is evaluated and then converted into integer format. This integer value then points to a position in the specified list of line numbers. The line number in that position is the start of the subroutine to which program control will be transferred. Upon execution of a RETURN command, control will be transferred to the next line following the ON...GOSUB command.

If the integer value is less than or equal to zero or greater than the number of line numbers in the list of line numbers, an "ERROR - FC" will occur. All non-integer numbers are rounded to the nearest integer.

ON...GOSUB

Examples:

1.) **ON A GOSUB 10,20,30**

If A is equal to 1, 2, or 3, program control will be transferred to line 10, 20, or 30, respectively. If A is less than or equal to zero or is greater than 3, an "ERROR - FC" will occur.

2.) **ON Z/D GOSUB 10,20,30**

If the integer value of Z divided by the value of D is equal to 1,2 or 3, program control will be transferred to the subroutine starting at line number 10, 20 or 30, respectively.

ON...GOTO

Format:
```
ON<num exp> GOTO <line num list>
```

where:

num exp = A numeric constant, numeric variable, or numeric expression.

line num list = A list of line numbers in a program separated by commas.

Purpose:

To transfer program control to one of a group of line numbers based on the evaluation of a numeric expression.

Description:

The ON...GOTO command is similar to the ON...GOSUB command, except that control is not passed to a subroutine, but transferred to a line number.

Example:

1.) **ON A GOTO 10,20,30**

If A is equal to 1, 2, or 3, program control will be transferred to line 10, 20, or 30, respectively. If A is less than or equal to zero or is greater than 3, an "ERROR - FC" will occur.

2.) **ON Z/D GOTO 10,20,30**

If the integer value of Z divided by the value of D is equal to 1, 2 or 3, program control will be transferred to line number 10, 20, or 30, respectively.

OPEN

Format:

```
                                     { <file name>  }
                   { [(<drive>)]     { *<trk>,<sec> }
OPEN#<ref>TO       { +[&<sec/buff>[,D]]

[,<char/rec>][,W][&<sec/buff>[,D]] }
                                   [;#...]
```

where:

ref = 2 for the modem device, or a file reference number (4-24) which may be a numeric constant or numeric variable.

drive = 1 for the upper drive or 2 for the lower drive; if this option is not specified, it defaults to 1.

file name = The name of a disk file which may be only a character constant.

trk = A track on the diskette which may be a numeric constant or a numeric variable.

sec = A sector within a track on the diskette which may be a numeric constant or a numeric variable.

char/rec = The number of characters per record which may be a numeric constant, numeric variable, or numeric expression; if this option is not specified, it defaults to 128 if a file name or a track and sector are specified in this command.

W = If the disk file is "Write-Protected", no error will occur when outputting to the file.

sec/buff = The number of sectors to be allocated to the file reference number which may be a numeric constant, numeric variable, or numeric expression; if this option is not specified, it defaults to 1.

D = Double buffered.

Purpose:

To allocate space in the File Buffer for a file reference number.

Description:

When this command is executed and only the file reference number and the number of sectors per buffer are specified, such as:

```
OPEN #4 TO+&3,D
```

the specified amount of space in the File Buffer (plus 35 bytes for a file header) will be allocated for the file reference number. In the example above, 6 sectors (3 sectors double buffered) plus 35 bytes are reserved in the File Buffer for file reference number 4 and can be illustrated as:

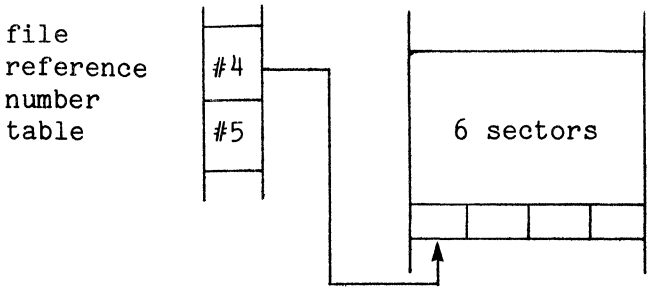


FIGURE #7-12

When the OPEN command consists of a file name, the disk file will then be associated with the file reference number. This will be recorded into the file header related to the file reference number, along with the other parameters specified in the command. For example:

```
OPEN #5 TO "TEST",80,W&2
```

reserves two sectors of File Buffer space for file reference number 5, associates disk file "TEST" with the reference number and records all of the parameters corresponding to the reference number in the file header. This can be illustrated as:

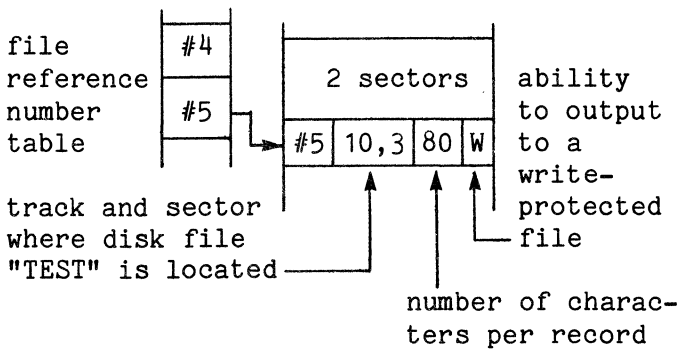


FIGURE #7-13

When an OPEN command specifies a track and sector, everything is performed as described above using a track and sector instead of a disk file. OPENing a track and sector to a file reference number gives the programmer the ability to access that track and sector through the end (Track 76, Sector 26) of the diskette.

OPEN

Note: No character variables or character arrays may be used prior to the execution of an open command.

Please refer to Chapter 5 for a more de-Tailed description of this command.

Examples:

1.) OPEN #4 TO+&5

Allocates 5 sectors plus a file header for file reference number 4 in the File Buffer.

2.) OPEN #4 TO+&5;#5 TO "TEST",50

Similar to the example above, but also assigns disk file "TEST" to file reference number 5 with 50 characters per record, and allocates 1 sector plus a file header in the File Buf-fer.

3.) OPEN #I TO "TEST",K&XYZ

Allocates the value of XYZ sectors plus a file header for the file re-ference number whose value is I and assigns it to disk file "TEST" having the value of K number of characters per record.

4.) OPEN #4 TO *1,3;#I TO "TEST",W&2,D

Allocates one sector and a file header for reference number 4 and assigns it to the third sector of the first track, and allocates 4 sectors (2 sectors, double buffered) plus a file header for the reference number whose value is I and assigns it to the disk file "TEST". Outputs can be directed to this disk file if it is Write-protected.

OPTION BASE

Format: OPTION BASE <num exp>

where: num exp = a numeric expression which is evaluated to 0 or 1.

Purpose:

To change the base of all dimensions in all arrays.

Description:

As stated previously, the base of all subscripts starts at zero. For example, DIM A(10) actually reserves 11 elements for array A; A(0),...A(10). If the pro-grammer wishes to change the base of the subscripts to 1, the OPTION BASE command is used and the logical expression must be true. Whenever the specified expression evaluates to 0, the base of all subscripts will be zero.

For example, if the following com-mands are executed:

```
10 OPTION BASE 1
20 DIM X(100)
```

array X will contain 100 elements; X(1),...,X(100). However, if the follow-ing commands are executed:

```
10 DIM X(100)
20 OPTION BASE 1
```

array X will contain 101 elements; X(1),...,X(101).

Examples:

1.) OPTION BASE A

Changes the base to 1 or 0, depending on the value of A.

2.) OPTION BASE 1

Changes the base to 1.

3.) OPTION BASE 0

Changes the base to 0.

POKE

Format: POKE <address>,<val>

where: address = A memory address in decimal format which may be a numeric constant or a numeric variable.

val = A decimal value which may be a numeric constant or a numeric variable.

Purpose:

To change the contents of a location in memory.

Description:

When this command is executed, Comm-Stor determines if the specified address is a valid location. If it is, Comm-Stor changes the location to the specified value.

If the value is less than zero or greater than 255, an "ERROR - FC" will occur.

Examples:

1.) POKE 1,60

Places 60 into location 1.

2.) POKE A,B

Places the value of B into the location which is the value of A.

3.) POKE DEC("838"),0

Places zero in the location referenced by the hexadecimal value 838.

4.) POKE A+Z,C/5

Places the value of C divided by 5 into the location referenced by the value of A plus the value of Z.

POSITION

Format: POSITION #<ref>,<rec>

where: ref = A file reference number (4-24) which is associated with a disk file that may be a numeric constant or a numeric variable.

rec = A record number within the disk file which may be a numeric constant, numeric variable, or a numeric expression.

Purpose:

To reposition the pointer to the beginning of a record in a disk file and to set the boundaries for the file to the boundary of that record.

Description:

The POSITION command is used in conjunction with the sequential READ # and PRINT # commands. Its purpose is to reposition the pointer to a logical record so that a READ # or PRINT # command (with no record numbers specified) may be executed.

When a POSITION command is executed, the boundaries of the disk file are changed to the boundaries of the logical record. If the programmer attempts to output data to or input data from the file outside the boundaries, an "ERROR - EF" will occur.

After a POSITION command is executed, followed by a sequential PRINT # command, the data will be output in a sequential manner. If another POSITION command is

POSITION

then executed, the previous POSITIONed record will be filled with nulls following the data already there.

Examples:

1.) **POSITION #4,5**

Repositions the pointer to the first position in the fifth logical record of the disk file associated with reference number 4.

2.) **POSITION #I,J**

Similar to the above example, except the record number is the value of J and the file reference number is the value of I.

PRESS

Format:

PRESS[(<drive>)]<file name>[,R][,<range>]

where: drive = 1 for the upper drive or 2 for the lower drive; if this option is not specified, it defaults to 1.

file name = The name of a disk file saved in text format which may be a character constant or a character variable.

R = RUN

range = A range of line numbers as described in the LIST command.

Purpose:

To enter a BASIC program into the Program Text Buffer without any REM commands or spaces.

PRINT

Description:

This command works similarly to the MERGE command except that any BASIC Lines consisting of a REM command and any spaces which appear in the program are deleted from the program before it is loaded into the Program Text Buffer. This can be useful if the program does not quite fit into the User Memory Space in its original form.

Examples:

1.) **PRESS "PROG 1"**

Merges "PROG 1" into the Program Text Buffer without any REM commands or spaces.

2.) **PRESS (2) A$,10-100**

Merges lines 10 through 100 from the file on the lower drive whose name is contained in A$, and deletes any REM commands and spaces from those lines.

PRINT

Format: PRINT [<list>[;]]

where: list = A list of constants and/or variables separated by commas or semi-colons.

Purpose:

To display values of characters to the invoker device.

Description:

When a PRINT command is executed, the specified values or characters will be displayed to the invoker device (terminal or modem port). If the parameters in the list are separated by commas, the display will be tabulated across a line on the terminal. If the parameters in the list are separated by semi-colons, no spaces (except those for the sign of a number)

PRINT

will appear in the display. When a semi-colon follows the list, a [RETURN] will not be issued to the terminal device after the values are displayed.

Specifying a PRINT command alone will cause a [RETURN] to be issued to the invoker device.

Examples (with the terminal as the invoker):

1.) **PRINT A,C**

Displays the values of A and C tabulated on the terminal, and issues a [RETURN] to the terminal.

2.) **PRINT A,C;**

Similar to example #1, except a [RETURN] is not issued to the terminal after the value of C is displayed.

3.) **PRINT "A$=";A$**

Displays the characters "A$=" immediately followed by the contents of A$ and a [RETURN].

4.) **PRINT A+1,B/C;"*****";**

Displays the value of A plus 1, tabs to the next field, and displays the value of B divided by the value of C immediately followed by four asterisks.

PRINT USING

Format: **PRINT USING <image>;[<list>[;]]**

where: image = The line number of an IMAGE command or a character constant which is an image format or a character variable containing an image format.

list = A list of constants and/or variables separated by commas or semi-colons.

PRINT USING

Purpose:

To display values of characters to the invoker device according to a specified format.

Description:

The PRINT USING command may be employed in situations where a specific output format is desired. The display is controlled by the image used and by the commas or semi-colons in the list (in the same manner as the PRINT command).

The following is a list of the options available in an image and their brief descriptions. Please refer to Chapter 4 for a more detailed explanation of this command.

Option	Description
#	one character position
#.#	round the least significant digit
T#.#	truncate the least significant digit
+#	+ if positive, - if negative
#+	same as above
-#	space if positive, - if negative
#-	same as above
E#.#	exponential notation
$#.#	dollar sign
$$$.#	floating dollar sign
#,###	insert a comma
##I(<chars>)##	insert any characters
S	space
N	null
B	binary mode
D	configured delimiter
A(<val>)D	temporarily change the delimiter to ASC<xx>
;	field separator
P(<val>)	random access for PRINT # USING
X	used for READ # USING

PRINT

Format:

PRINT #<ref>[,<rec>][&#<ref>[,<rec>]

[&#...]];[<list>[;]]

where: ref = A port reference number (0-3) or a file reference number (4-24) which is associated with a disk file that may be a numeric constant or a numeric variable.

rec = A logical record number within a disk file which may be a numeric constant or a numeric variable.

list = A list of constants and/or variables separated by commas or semi-colons.

Purpose:

To output data to a device or a disk file.

Description:

More than one port reference number is allowed in this command, but if more than one file reference number is specified, output will only be directed to the disk file associated with the right-most file reference number.

This command is similar to the PRINT command except that the output may be directed to any device or file. Additionally, a [RETURN] character will follow each different value or character string when the output is directed to a disk file.

If a record number accompanies a file reference number in this command, the pointer in the disk file will be repositioned to the first position in that record and the boundaries of the file will be changed to the boundaries of that logical record.

Examples:

1.) PRINT #0&#4;"ABC"

Outputs the characters "ABC" (followed by a [RETURN]) to the device which issued the .BA command and to the file associated with reference number 4.

2.) PRINT #I;A,B,C

Outputs the values of A,B, and C (each followed by a [RETURN]) to a port or disk file associated with the reference number whose value is I.

3.) PRINT #I,J;A$

Outputs the contents of A$, followed by a [RETURN], into the J record of the disk file associated with the reference number whose value is I.

PRINT # USING

Format:

PRINT #<ref>[,<rec>][[&#<ref>[,<rec>]

[&#...]] USING<image>;[<list>[;]]

where: ref = A port reference number (0-3) or a file reference number (4-24) which is associated with a disk file which may be a numeric constant or a numeric variable.

rec = A logical record number within a disk file which may be a numeric constant or a numeric variable.

image = A line number of an IMAGE command or a character constant which is an image format or a character variable containing an image format.

PRINT # USING

list = A list of constants and/or variables separated by commas or semi-colons.

Purpose:

To output data to a device or a disk file using a specified format.

Description:

This command is a combination of the PRINT # and PRINT USING commands. Operations proceed as described for the PRINT USING command except that the output is sent to a device or a disk file. Please refer to Chapter 4 for a more detailed explanation.

Examples:

1.) **PRINT #0&#4 USING 10;"ABC"**

Outputs the characters "ABC" to the device which issued the .BA command and the disk file associated with file reference number 4 using the format specified in the IMAGE command in line 10.

2.) **PRINT #I,5 USING IM$;A,B,C**

Outputs the values of A,B, and C into the fifth record of the disk file associated with the reference number whose value is I using the image contained in IM$.

3.) **PRINT #2 USING "10#D";A$**

Outputs to the modem device the contents of A$ using 10 character positions followed by a [RETURN].

RANDOMIZE

Format: RANDOMIZE

Purpose:

To generate a new sequence of random numbers.

Description:

When the RND function is used without any arguments, this command will generate a new sequence of random numbers for that function.

Example: RANDOMIZE

READ

Format: READ <var list>

where: var list = A list of variables separated by commas.

Purpose:

To assign values from DATA commands to variables.

Description:

The READ command assigns values associated with the internal data pointer to the specified variables in their respective order. The internal data pointer is associated with values specified in DATA commands.

A pointer is positioned at the first value in the first DATA command when the first READ command is executed. The indicated value is assigned to a variable and the pointer is moved to the next value in the DATA command.

Subsequent READ commands take values from the location at which the data pointer is positioned. The pointer moves sequentially through the DATA commands unless it is repositioned by a RESTORE command.

If all the DATA commands have been accessed and a READ command is executed, an "ERROR - OD" will occur.

READ

When the pointer is positioned at a numeric value and a character variable is specified in the variable list, the numeric value will be converted to a character string and an attempt will be made to READ a numeric variable; however, an "ERROR - TM" will occur.

Examples:

1.) **READ A,B$**

Obtain a numeric value from the location of the pointer and assign it to A; obtain a character string and assign it to B$.

2. **READ A(N),A$(A(N))**

Obtains the numeric value for A(N) and uses this value as a reference to an element in array A$ and obtains the value for this element.

READ

Format: READ #<ref>[,<rec>];<var list>

where:
- ref = A port reference number (0-3) or a file reference number (4-24), associated with a disk file which may be a numeric constant or a numeric variable.
- rec = A logical record number within a disk file which may be a numeric constant or a numeric variable.
- var list = A list of variables separated by commas.

Purpose:

To obtain data from a device or a disk file.

Description:

If the reference number specified is a file reference number, Comm-Stor will obtain a data item from the disk file associated with that reference number and assign it to the specified variable. Each data item in a disk file should be terminated by a [RETURN] character so that the data may be obtained correctly. If there is no [RETURN] character following each data item and two or more character variables are specified in the variable list, the first 255 characters will be assigned to the first character variable, the next 255 characters will be assigned to the next character variable, and so on. Specifying a record number in this command will cause the boundaries of the disk file to be changed to the boundaries of the logical record.

If a port reference number is specified, Comm-Stor will wait until data is entered from the specified device in order to assign that data to the specified variable(s).

Examples:

1.) **READ #2;A$**

Obtains a character string from the modem device and assigns it to A$.

2.) **READ #4,3;A$**

Similar to example #1 except that the characters are obtained from the third record of the disk file associated with reference number 4.

3.) **READ #1;X,Y,Z**

Obtains three values from the terminal and assigns them to X,Y, and Z.

4.) READ #I;X,Y,Z

Obtains three values from the device or disk file associated with the reference number whose value is I and assigns these values to X,Y, and Z.

READ # USING

Format:

READ #<ref>[,<rec>]USING<image>;<var list>

where: ref = A port reference number (0-3) or a file reference number (4-24) assigned to a disk file which may be a numeric constant or a numeric variable.

rec = A logical record number within a disk file which may be a numeric constant or a numeric variable.

image = A line number of an IMAGE command, or a character constant which is an image format, or a character variable containing an image format.

var list = A list of variables separated by commas.

Purpose:

To obtain data from a device or a disk file, using a specified format.

Description:

Values are obtained for the specified variables for this command just as for the READ # command except that the specified format indicates how the values will be obtained.

Please refer to Chapter 4 for a more detailed description of this command.

The following is a list of the options available in an image and their brief descriptions.

Option	Description
#	one character position
#.#	three character positions
T#.#	same as above
+#	two character positions
-#	same as above
#+	same as above
#-	same as above
$#	same as above
#,###	four character positions
##I(AB)##	six character positions
S	ignore next character position
N	same as above
B	binary mode
D	acknowledge the configured delimiter
A(xx)	acknowledge the temporarily changed delimiter
;	field separator
P(xx)	reposition within a record
X	do not echo characters entered from the terminal device

Examples:

1.) READ #2 USING 10;A$

Obtains characters according to the IMAGE command in line 10 and assign them to A$ from the modem device.

2.) READ #4,3 USING 10;A$

Similar to the above example, but the characters are obtained from the third record in the disk file associated with reference number 4.

3.) READ #1 USING "10#";A$

Obtains 10 characters from the terminal and assigns them to A$.

4.) READ #0 USING IM$;A,B

Obtains two values from the invoker device according to the image contained in IM$ and assigns them to A and B.

RECSIZE

Format: RECSIZE #<ref> TO <num exp>

where: ref = A file reference number (4-24) associated with a disk file which may be a numeric constant or a numeric variable.

num exp = A numeric constant, a numeric variable, or a numeric expression.

Purpose:

To change the number of characters per record associated with a file reference number.

Description:

As stated in the ASSIGN and OPEN commands, the number of characters per record associated with a disk file and a file reference number is recorded in the file header. To change this parameter, the RECSIZE command may be used. When the RECSIZE command is executed, the numerical expression is evaluated; this value is recorded in the file header associated with the specified file reference number.

Examples:

1.) RECSIZE #4 TO 80

Changes the number of characters per record for file reference number 4 to 80.

2.) RECSIZE #I TO J

Similar to the example above except that the reference number is the value of I and the number of characters per record is the value of J.

3.) RECSIZE #I TO LEN(B$)

Changes the record size for the reference number whose value is I to the number of characters in B$.

RELEASE

Format: RELEASE [#<ref>][,T][;#...]]

where: ref = A file reference number (4-24) associated with a disk file which may be a numeric constant or numeric variable.

T = Truncate

Purpose:

To deactivate a disk file from a BASIC program and to release a portion of the File Buffer.

Description:

Issuing a RELEASE command causes Comm-Stor to deactivate a disk file (referenced by an OPEN or ASSIGN command) from a BASIC program. Additionally, the area in the File Buffer related to the specified reference number will no longer be associated with that reference number. That is, the RELEASEd File Buffer space may be used with a new file reference number or several reference numbers, depending on the amount of space RELEASEd and the buffer size allocated to each new reference number by an ASSIGN command.

As discussed in Chapter 5, a 35-byte file header is reserved in the File Buffer along with the number of sectors specified in the OPEN or ASSIGN command. This file

RELEASE

header does not need to be considered if every disk file reference number used in the BASIC program is specified in an OPEN command; it should only be considered when RELEASE is used when other file reference numbers utilize the resultant available space in the File Buffer. This is illustrated in the following example:

```
10 CREATE "TEST1"; "TEST2"; "TEST3"
20 OPEN #4TO"TEST1";#5TO"TEST2";
#6TO"TEST3"
.
.
.
.
```

The example below shows how the file header must be considered:

```
10 OPEN #4TO+&4
20 RELEASE #4
30 CREATE "TEST1";"TEST2";"TEST3"
40 ASSIGN #4TO"TEST1";#5TO"TEST2";
#6TO"TEST3"
```

If only three sectors were allocated in the File Buffer in line 10 above, an "ERROR - FO" would occur since three sectors plus two extra file headers are needed for the file reference numbers 4, 5, and 6, in line 40.

If a "T" is specified in the RELEASE command, any unused sectors within the file(s) will be released for further file space on the diskette. This will happen immediately if the file is the last Directory entry on the disk. If it is not, the file will be truncated when a Comm-Stor SQUISH (.SQ) command is executed.

Examples:

1.) **RELEASE**

Closes all files and releases the File Buffer space used by their associated file reference numbers.

2.) **RELEASE #4**

Closes and releases file reference number 4.

3.) **RELEASE #5;#4;#I**

Closes and releases file reference numbers 4, 5 and the value of I.

4.) **RELEASE #4;#5,T**

Closes and releases file reference numbers 4 and 5 and truncates the disk file which was associated with reference number 5.

Sample Program:

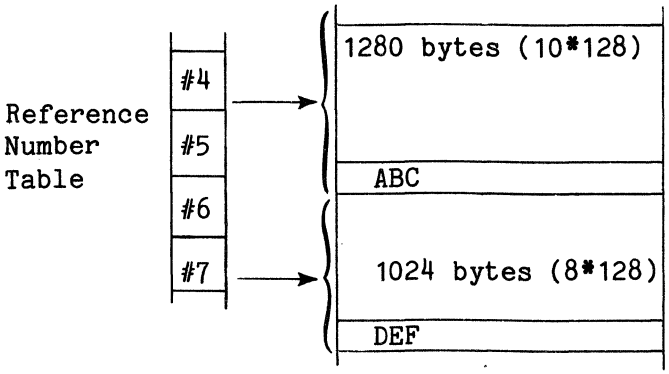
```
 10 CREATE "BOB",128*100;"ROY",128*100
 20 OPEN #4 TO "ROY"&10
 30 FOR I=1 TO 128*50
 40 PRINT #4;USING"#";"A"
 50 NEXT
 60 RELEASE #4,T
 70 ASSIGN #4 TO"ROY"&5;#5TO"BOB"&4
 80 FOR I=1 TO 128*50
 90 READ #4 USING "#";A$
100 PRINT #5 USING "#";A$
110 NEXT
120 CLOSE #5,T
130 END
```

Two files, "BOB" and "ROY", are created to contain 12,800 characters (each) in line 10. "ROY" is then OPENed with a buffer size of 10 sectors and the character "A" is written 6,400 times to "ROY". "ROY" is then RELEASEd in line 60 (which frees its 10 sector buffer plus its file header) and is truncated, reducing its file size from 12,800 characters to 6,400 characters. "ROY" is ASSIGNed to file reference number 4 using 5 sectors plus 35 bytes of File Buffer space. One character is then obtained from "ROY" and is written to "BOB" 6,400 times. File reference number 5 is CLOSEd and truncated (in line 120), reducing the size of "BOB" from 12,800 characters to 6,400 characters. (When execution of the program is complete, file reference number 4 will be au-

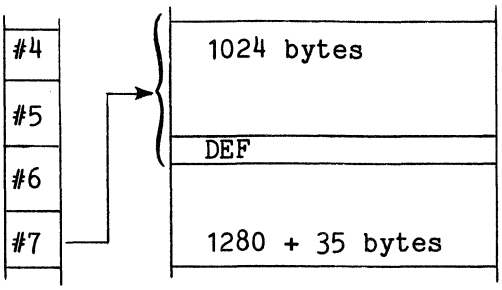
tomatically CLOSEd.) This is illustrated in Figure #7-14 below.

EXAMPLE OF FILE BUFFER ALLOCATION

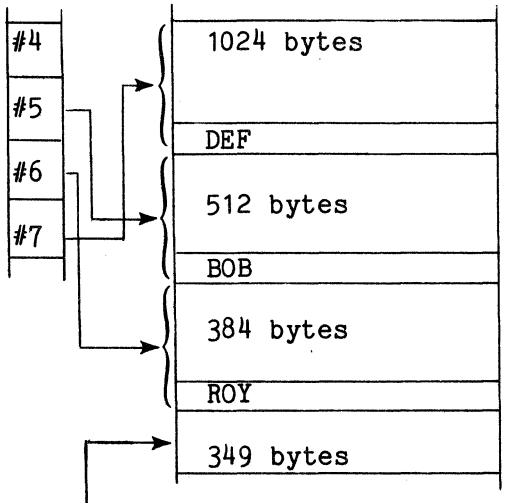
OPEN #4 TO "ABC"&10;#7 TO "DEF"&8



RELEASE #4



ASSIGN #5 TO "BOB"&4; #6 TO "ROY" &3



349 bytes of available File Buffer space remain and is calculated by:

1280+35-(512+35)-(384+35)=349

FIGURE #7-14

REM

Format: REM [<comment>]

where: comment = A group of characters which comment on the operations of a program.

Purpose:

To insert a comment or description into a program.

Description:

The REM command permits comments to be inserted into a program to clarify and improve readability. It is never executed, but program control may be transferred to it. The command is terminated by a [RETURN] character rather than a colon.

For example, the following line will not cause P to equal 1.5.

```
10 REM SET UP PERCENT:P=1.5
```

The following line will cause P to equal 1.5.

```
10 P=1.5:REM SET UP PERCENT
```

Examples:

1.) REM

2.) REM THIS PROGRAM COMPUTES COMPOUND INTEREST

RENUM

Format: RENUM[<line num>][,<inc>]

where: line num = The line number in the first line of the current program, which may only be a numeric constant.

RENUM

inc = The increment used for each successive line number which may only be a numeric constant.

Purpose:

To renumber the line numbers of the current program.

Description:

The following conditions apply to the RENUM command:

1.) If this command is entered alone, the first line number in memory will be 100. Each successive line number will then be incremented by 10.

2.) If a line number is specified in this command, the first line in the Program Text Buffer will have that line number associated with it. If an increment is also specified, each successive line number will be increased by this increment.

3.) If just a line number is specified, the increment is assumed to be 10.

4.) If just an increment is specified, the first line number is assumed to be 100.

5.) Whenever the line numbers are renumbered, all references to line numbers except the characters in character constants and REM commands will also be changed. If an attempt is made to change a line number when that line number is nonexistent, asterisks will be printed in place of the line number.

Examples:

1.) **RENUM**

RENUMber all line numbers starting with 100 and incremented by 10.

2.) **RENUM 1**

Similar to the above example, but the first line number is 1.

3.) **RENUM,5**

RENUMber all lines starting at 100 and incremented by 5.

4.) **RENUM1,5**

RENUMber all lines starting at 1 and incremented by 5.

RESTORE

Format: RESTORE [<line num>]

where: line num = A line number in the current program.

Purpose:

To reset the pointer to the beginning or to another place in the internal data table.

Description:

If this command is entered alone, the pointer will be repositioned to the beginning of the internal data table allowing the data items to be read again.

If a line number is specified in this command, the pointer will be repositioned to the first data item specified in the DATA command associated with the specified line number.

Examples:

1.) **RESTORE**

Repositions the pointer to the beginning of the internal data table.

RESTORE

2.) RESTORE 100

Repositions the pointer to the first data item in line number 100.

RETURN

Format: RETURN

Purpose:

To terminate execution of a subroutine, a CALLed program, or an RTCS interrupted program.

Description:

The RETURN command is the last command to be executed in a subroutine, a CALLed program, or an RTCS interrupted program.

There may be more than one RETURN command, but the first one encountered during execution of the program terminates the routine and returns control to the line number following the last executed GOSUB, ON...GOSUB, CALL, or the line following the line where the RTCS interrupt occurred.

REWIND

Format: REWIND #<ref>

where: ref = A file reference number (4-24) which is associated with a disk file; it may be a numeric constant or a numeric variable.

Purpose:

To start the accessing of a disk file at the first character position in the file.

Description:

When this command is executed, the pointer related to the disk file associated with the specified file reference number will be repositioned to the beginning of the disk file. The second boundary of the disk file will be repositioned to the last character position in the file.

Please refer to Chapter 5 for a more detailed explanation of this command.

Examples:

1.) REWIND #4

Rewinds the disk file associated with reference number 4.

2.) REWIND #I

Rewinds the disk file associated with the reference number whose value is I.

RTCS

Format: RTCS<log exp>

where: log exp = A logical expression.

Purpose:

To enable or disable the Real-Time Control System.

Description:

When this command is executed and the logical expression is evaluated to be true (any value but 0), the Real-Time Control System will be enabled. In other words, detection of all interrupts specified by the programmer will be turned on.

If the logical expression is evaluated to be false (0), all interrupts will be disabled. Program detection of interrupts will not be active.

Please refer to Chapter 6 for more information on the Real-Time Control System.

RTCS

Examples:

1.) RTCS 1

Enables the Real-Time Control System.

2.) RTCS 0

Disables the Real-Time Control System.

3.) RTCS A

Depending on the value of A, enables or disables the Real-Time Control System.

4.) RTCS NOT A

Depending on the value of NOT A, enables or disables the Real-Time Control System. If A=0, the system will be enabled; any other value will disable the system.

RUN

Format: RUN[<line num>]

where: line num = A line number in a program.

Purpose:

To execute the program which resides in the Program Text Buffer.

Description:

When a RUN command is executed, a CLEAR operation is automatically executed. If a line number is specified in this command, execution will start at that line number. If a line number is not specified, execution will begin at the lowest numbered line number.

Program execution can be interrupted by entering a [CNTRL/T]; all variables return to a value of zero.

SAVE

A program may be re-executed without performing the CLEAR operation by entering GOTO with a line number.

Examples:

1.) RUN

2.) RUN 100

SAVE

Format:

SAVE[(<drive>)][<file name>[,T][,<range>

]][;[(<drive>)]...]

where: drive = 1 for the upper drive or 2 for the lower drive; if this option is not specified, it defaults to 1.

file name = The name of a disk file which will contain the current program; it may be a character constant or a character variable.

T = Text format; if this option is omitted, it defaults to packed format.

range = A range of line numbers as described in the LIST command.

Purpose:

To create a disk file and store the current program into it.

Description:

Upon execution of the SAVE command, the diskette inserted in the specified drive is searched to find a file whose name is the same as the specified file

name. If one is found while in DIRECT MODE, "--SURE?" will be displayed on the terminal. A programmer response of "Y" (for yes) will execute the SAVE command and, consequently, will destroy all of the previous contents of that disk file and enter the current lines into that file. A response of "N" (for no) will not execute the SAVE command, at which time the programmer may choose a different file name to save the program.

A "T" specified in the SAVE command will store the program in text format. This enables the future use of a MERGE and/or PRESS command to be issued on that file and a Comm-Stor command of .D will display the file in an understandable format. If the program is saved in text format, "TXT" will appear to the right of the sector byte count in a Directory display, (.DD * or .DS L). If a "T" is not specified in this command, the program will be saved in packed format and "PAK" will appear to the right of the sector byte count.

If a range of line numbers is specified in this command, only those line numbers which are included in the specified range will be entered into the disk file.

If this command is issued without any parameters, the program will be saved in its original format and in the same disk file as it was originally. If the disk file was a standard ASCII file, an "ERROR - NO FIND" will occur.

When a SAVE command is executed, the total number of characters including spaces which reside in the Program Text Buffer, is calculated by the system. This figure is known as the ACTUAL BYTE COUNT and is the first number which appears after the file name (or extension) in a Directory display. Comm-Stor then calculates the number of sectors (128 characters each) needed to hold the actual byte count. This figure is called the SECTOR BYTE COUNT and is found to the right of the actual byte count in a long Directory display. For example, if the actual byte count of a file is 2548, the sector byte count will be 2560 because 2548 divided by 128 equals 19.9 and 128 multiplied by 20 equals 2560. The actual and sector byte counts are determined in this manner for every disk file, not only those created by a SAVE command.

Examples:

1.) **SAVE "ABC"**

Stores the current program in the disk file "ABC" in packed format.

2.) **SAVE "ABC",T,10-100**

Stores lines 10 through 100 of the current program in disk file "ABC" in text format.

3.) **SAVE A$**

Saves the current program in the disk file whose name is contained in A$ and is in packed format.

4.) **SAVE(2)A$,T**

Saves the current program, in text format, in the disk file whose name is contained in A$ located on the lower drive.

SET

Format: SET{$! % #}<range>[;{$! % #}<range>[;...]]

where: $ = Character variables.

! = Single precision variables.

% = Integer variables.

= Extended precision variables.

range = 1 or 2 letters (A-Z) separated by a hyphen.

Purpose:

To declare variables as specified types.

Description:

Upon execution of a SET command, any variables starting with the specified range of letters will be declared as a specified type of variable. For example, if

SET $E-G

is specified, any variables whose names begin with an "E", "F", or "G" will be character variables. If the type of a variable is to be changed, another SET command may be issued.

Examples:

1.) **SET %I**

Declares any variables whose names start with an "I" to be integer variables.

2.) **SET $A;!B;%C;#D**

Declares all variables beginning with the letter "A", "B", "C", and "D" to be character, single precision, integer, and extended precision variables, respectively.

Note: The programmer may also change the type of a variable by specifying $, !, #, or % after a variable name. However, when the variable name alone is specified, its type remains the same as specified in the SET command (or default = single precision). For example, after SET #A is executed and A%=1 is executed, A and A% are not the same variable.

SETIME

Format: SETIME #<code> TO <char exp>

where: code = 0 for the actual time, 1 for TIMER 1, or 2 for TIMER 2; this option may be a numeric constant or a numeric variable.

char exp = A character expression consisting of 6 numeric characters that are in 24 hour format.

Purpose:

To enter the time of day into the Comm-Stor system.

Description:

When this command is executed, the character expression is evaluated and the time is entered into the system. 24 hour time is used (i.e., twelve midnight is "000000" and one second before midnight is "235959").

These times can be used with the Real-Time Clock functions and with WHEN commands.

Please refer to Chapter 6 for a more detailed description of this command.

Examples:

1.) **SETIME #0 TO "001000"**

Enters 10 minutes past midnight as the actual time.

2.) **SETIME #1 TO "120000"**

Enters 12 noon into TIMER1.

3.) **SETIME #2 TO A$**

Enters the time contained in A$ into TIMER2.

SETIME

4.) SETIME #I to A$

Enters the time contained in A$ into the current time or one of the TIMERs, depending on the value of I.

STOP

Format: STOP

Purpose:

To interrupt execution of a program.

Description:

Upon execution of the STOP command, the program in progress is interrupted and the message "BREAK", followed by the line number where the BREAK occurred, is displayed on the terminal. This is useful for debugging purposes, such as determining the value of a variable at a particular moment within a program.

To resume execution of the program, a CONT command should be used. Please see the description of the CONT command.

Example:

35 STOP

SYSTEM

Format: SYSTEM [(<drive>)] <file name>

where: drive = 1 for the upper drive or 2 for the lower drive; if this option is not specified, it defaults to 1.

file name = The name of a disk file containing an object program which may be a character constant or a character variable.

TRACE

Purpose:

To load an object program into the Object Buffer.

Description:

When the SYSTEM command is executed, the object code residing in the specified disk file is loaded into the address specified by the first two bytes of the object code being loaded (normally the Object Buffer).

The object program is specially produced by Sykes Datatronics, Inc., and is accessed by the BASIC language using the USC command and USF function.

Examples:

1.) SYSTEM "PROG 1"

Loads object program (in disk file "PROG 1") into the Object Buffer.

2.) SYSTEM (2) A$

Loads the object program whose file name is contained in A$, which exists on the lower drive of a dual drive unit into the Object Buffer.

3.) SYSTEM LEFT$(A$,10)

Loads the object program (in the disk file whose name is the first 10 characters in A$) into the Object Buffer.

TRACE

Format:

TRACE[#<ref>[&#<ref>[&#...]];]<log exp>

where: ref = A port reference number (0-3) or a file reference number (4-24) which is associated with a disk file; it may be a numeric constant or a numeric variable.

TRACE

log exp = A logical expression.

Purpose:

To trace execution of a program.

Description:

Upon execution of the TRACE command, the logical expression is evaluated. If it is false (0), tracing of the program will not occur. Otherwise, tracing will take place.

Tracing enables the programmer to see which lines in the current program are being executed; this is extremely helpful for debugging purposes. The order of the executed lines is shown by displaying their corresponding line numbers surrounded by brackets, []. Any PRINT commands will cause the data to be displayed on the same line as the bracketed line number.

If reference numbers are specified, the display of line numbers will be output to a device or a disk file. If more than one file reference number is specified, the output will be directed to the disk file associated with the last file reference number specified in the command.

The TRACE command may be entered into the system in Direct Mode or in RUN mode. If it is executed within a program, tracing will begin at the next line following the TRACE command.

Examples:

1.) **TRACE 1**

Traces the execution of the current program.

2.) **TRACE #2&#4;1**

Traces the current program and displays it to the modem and the disk file associated with file reference number 4.

3.) **TRACE A**

Depending on the value of A, enables or disables the tracing of the execution of the program.

4.) **TRACE #3;A**

Depending on the value of A, enables or disables the tracing of the execution of the program on the printer.

TRUNCATE

Format:

TRUNCATE[(<drive>)]<file name>

[;[(<drive>)]...]

where: drive = 1 for the upper drive or 2 for the lower drive; if this option is not specified, it defaults to 1.

file name = The name of a disk file, which may be a character constant or a character variable.

Purpose:

To delete any unused disk space from a disk file.

Description:

Execution of this command removes any unused bytes from the specified disk file, if possible. If it is not possible at the time, the file will be truncated when a Comm-Stor SQUISH (.SQ) command is executed.

Examples:

1.) **TRUNCATE "TEST"**

Truncates disk file "TEST".

2.) **TRUNCATE (2) A$**

Truncates the disk file on the lower drive whose name is contained in A$.

3.) **TRUNCATE LEFT$(A$,10);"TEST"**

Truncates the disk file whose name is the first 10 characters in A$ and the disk file "TEST".

UNLOCK

Format: UNLOCK #<ref>

where: ref = A file reference number (4-24), which is associated with a disk file; it may be a numeric constant or a numeric variable.

Purpose:

To reposition the second boundary of a disk file to the end of the file.

Description:

The UNLOCK command is used when the boundaries of a disk file have been repositioned to a logical record in the file and when the second boundary of the file is to be changed to the end of the file (for sequential access).

Please refer to Chapter 5 for a more detailed explanation of this command.

Examples:

1.) **UNLOCK #4**

Unlocks the second boundary of the disk file associated with reference number 4.

2.) **UNLOCK #I**

Similar to the example above except that the reference number is the value of I.

USC

Format: USC<code>

where: code = A value between 0 and 127 which may be a numeric constant or a numeric variable.

Purpose:

To execute a command located in the Object Buffer.

Description:

After a SYSTEM command is executed, the object code is available to the BASIC language through the use of an USC command.

Example:

1.) USC 6

Executes the object code which corresponds to the code, 6.

2.) USC A

Executes the object code which corresponds to the value of A.

WAIT

Format: WAIT<address>,<val 1>[,<val 2>]

where: address = A location in memory which may be a numeric constant or a numeric variable.

val 1 = A numeric constant, numeric variable, or a numeric expression which is greater than or equal to zero and less than or equal to 255.

val 2 = Same as the description for "val 1".

Purpose:

To delay program execution until a location in memory contains an appropriate value.

Description:

When the WAIT command is executed, the value of "val 1" is ANDed with the value in the specified location. If the result is false (0) the program will delay. If it is true (not equal to zero) execution of the program will resume normally.

If the "val 2" option is specified in this command, its value is EXCLUSIVE ORed with the value in the specified location. This result is then ANDed with the value in "val 1" and the execution of the program will be exactly as described above.

Examples:

1.) **WAIT A,B**

Delays program execution until the value of B ANDed with the value in location A is true.

2.) **WAIT A,B,C**

Delays program execution until the value in location A EXCLUSIVE ORed with the value of C and ANDed with the value of B is true.

3.) **WAIT 16,16**

Delays program execution until 16 ANDed with the value in location 16 is true.

WATCH 1
WATCH 2

Format:

WATCH{1|2}<address>,<val 1>[,<val 2>]

where: address = A location in memory which may be a numeric constant or a numeric variable.

val 1 = A numeric constant, numeric variable, or a numeric expression which is greater than or equal to zero and less than or equal to 255.

val 2 = Same as the description for "val 1".

Purpose:

To prepare a hardware register for an interrupt condition.

Description:

The WATCH commands prepare the two hardware registers of the Real-Time Control System for the use of interrupts.

The operation of these commands is similar to that of the WAIT command, except a false value does <u>not</u> cause an interrupt to occur and a true value does cause an interrupt to occur.

Please refer to Chapter 6 for a more detailed explanation of this command.

Examples:

1.) **WATCH1 A,B**

Determines if the value of B ANDed with the value in location A is true.

WATCH 1
WATCH 2

2.) WATCH2 A,B,C

Determines if the value of C EXCLUSIVE ORed with value in location A and ANDed with the value of B is true.

3.) WATCH1 16,16

Determines if 16 ANDed with the value in location 16 is true.

WHEN

Format:

```
                 { GOSUB }
WHEN#<ref>       { GOTO  }
                 { END   }

<line num>[{&<priority>}[,E]{ }]
          {&E               }
```

where: ref = A reference number which corresponds to an interrupt condition, it may be a numeric constant or a numeric variable.

line num = A line number in a program which may be a numeric constant.

priority = A number signifying the priority of an interrupt condition; it may be a numeric constant or a numeric variable.

E = If specified, will give an ERROR if a double interrupt occurs.

Purpose:

To detect an interrupt and to specify what should happen when an interrupt occurs.

Description:

The WHEN command specifies that the current program will take proper action when an interrupt occurs. The WHEN command will only take effect if the Real-Time Control System is enabled. (See RTCS.)

If a priority is specified, the specified interrupt condition will have that priority associated with it. This is useful when multiple interrupt conditions occur at the same time.

If an "E" is specified in this command, an error will occur if the same interrupt occurs prior to the completion of a particular action for that interrupt condition.

Please refer to Chapter 6 for a more detailed explanation of this command.

Examples:

1.) WHEN #1 GOTO 500&7,E

This command specifies that program control should transfer to line 500 when a character is entered from the terminal. This detection is given highest priority of all WHEN commands. An error will result if another interrupt occurs at the same time as the terminal interrupt.

2.) WHEN #A GOSUB 1000

Depending on the value of A, this command specifies that the program should execute the subroutine starting at line 2000 when the interrupt occurs.

CHAPTER 8

FUNCTION DEFINITIONS

ABS

Format: ABS(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: The absolute value of "X".

Example: A = ABS(-5) A is equal to 5

ASC

Format: ASC(X$)

where: X$ = A character constant, a character variable, or a character expression.

Result: The decimal value corresponding to the first (left-most) ASCII character in "X$". (See Appendix C.)

Example: A = ASC("ABC") A is equal to 65

ATN

Format: ATN(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: The arctangent (in radians), in the range of - π/2 to π/2, of "X".

Example:

A = ATN(.5) A is equal to .4364761

CHR$

Format: CHR$(X)

where: X = A numeric constant, a numeric variable, or a numeric expression which is equal to or greater than 0 and less than or equal to 255.

Result: The ASCII character which is equivalent to the decimal value of "X".

Example: A$ = CHR$(65) A$ contains "A"

COS

Format: COS(X)

where: X = A numeric constant, a numeric variable, or a numeric expression which represents the measure of an angle in radians.

Result: The cosine of "X".

Example:

A = COS(1) A is equal to .54030231

CTIME

Format: CTIME(X$)

where: X$ = A character constant, a character variable, or a character expression which contains 6 numeric characters.

Result: The number of seconds in "X$" relative to midnight.

Example:

A = CTIME("051543") A is equal to 18943

CTIME$

Format: CTIME$(X)

where: X = A numeric constant, a numeric variable, or a numeric expression which represents a number of seconds.

Result: A character string containing 6 numeric characters which represents "X" as a time relative to midnight.

Example:

A$ = CTIME$(18943) A$ contains "051543"

DEC

Format: DEC(X$)

where: X$ = A character constant, a character variable, or a character expression consisting of 1 to 6 characters which represent a hexadecimal value.

Result: The decimal equivalent of the hexadecimal number contained in "X$".

Example: A = DEC("FF") A is equal to 255

EXP

Format: EXP(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: The natural exponent raised to the power of "X".

Example:

A = EXP(2) A is equal to 7.3890561

FILL

Format: FILL(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: When outputting to a disk file, this function will cause the ASCII equivalent of the decimal value of "X" (obtained from Appendix C) to be output to the disk file until the second boundary is reached. If the accessing of a disk file has been purely sequential, this function will FILL the remainder of the entire diskette file. If random access has been used, this function will FILL the remainder of the entire record.

FRE

Format: FRE(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: The number (in integer format) of bytes in the Free Space Buffer.

HEX$

Format: HEX$(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: A character string containing the hexadecimal representation of decimal value "X".

Example:

A$ = HEX$(31) A$ contains "1F"

INT

Format: INT(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: The integer which is the largest integer less than or equal to "X".

Example: A = INT(5.263) A is equal to 5

LEFT$

Format: LEFT$(X$,X)

where: X$ = A character constant, a character variable, or a character expression.

X = A numeric constant, a numeric variable, or a numeric expression which is greater than zero and less than or equal to 255.

Result: The left-most "X" characters in "X$".

Example:

A = LEFT$("ABCDEFG",3) A$ contains "ABC"

LEN

Format: LEN(X$)

where: X$ = A character constant, a character variable, or a character expression.

Result: The number of characters in "X$".

Example: A = LEN("ABC") A is equal to 3

LOG

Format: LOG(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: The natural logarithm of "X".

Example:

A = LOG(7.6) A is equal to 2.0281482

MID$

Format #1: MID$(X$,X)

Format #2: MID$(X$,X,Y)

where: X$ = A character constant, a character variable, or a character expression.

X = A numeric constant, a numeric variable, or a numeric expression which is greater than zero and less than or equal to 255.

Y = Same as the description for "X".

Result: If Format #1 is used, the characters in "X$" starting at position "X" are the result. If "X" is greater than the number of characters in "X$", the result will be a null string.

If Format #2 is used, the characters in "X$" starting at position "X" for "Y" characters are the result. If "Y" is greater than the number of characters in "X$", the system will ignore "Y" and treat the function as for Format #1.

MID$

Examples:

```
A$=MID$("ABCDEFG",4)    A$ contains "DEFG"
A$=MID$("ABCDEFG",4,2)  A$ contains "DE"
```

PEEK

Format: PEEK(X)

where: X = A numeric constant, a numeric variable, or a numeric expression which represents a location in memory.

Result: The value in location "X".

POS

Format: POS(X)

where: X = A port reference number (0-3) or a file reference number (4-24) which is associated with a disk file.

Result: If "X" is a port reference number, the result will be the current position of the pointer of that port.

If "X" is a file reference number and the disk file has been accessed only by sequential means, the result will be the last character position accessed. If no character has been accessed in the file, or if a REWIND command is executed, the result will be 1.

If the disk file has been accessed randomly, the result of this function will be the current record number.

Example:

```
PRINT TAB(5);"HELLO";
A = POS(1)            A is equal to 10
```

RIGHT$

Format: RIGHT$(X$,X)

where: X$ = A character constant, a character variable, or a character expression.

X = A numeric constant, a numeric variable, or a numeric expression which is greater than zero and less than or equal to 255.

Result: The right-most "X" characters in "X$".

Example:

```
A$=RIGHT$("ABCDEFG",3)   A$ contains "EFG"
```

RND

Format #1: RND

Result: A random number between 0 and 1, obtained from the sequence generated by the RANDOMIZE command.

Format #2: RND(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: If "X" is negative, a new sequence of random numbers is generated and the result will be the first random number in that sequence.

If "X" is positive, the result will be the next random number in the current sequence.

If "X" is zero, the result will be the last random number generated from the current sequence.

SENSE

Format: SENSE(X)

where: X = A numeric constant, a numeric variable, or a numeric expression which is equal to or greater than 1 and less than or equal to 17.

Result: -1 if condition "X" is true, or 0 if it is false.

Below are the corresponding conditions for "X":

```
 1 = TERMINAL CHARACTER PRESENT?
 2 = MODEM CHARACTER PRESENT?
 3 = DRIVE 1 READY?
 4 = DRIVE 1 VARIABLE LENGTH DISK?
 5 = DRIVE 1 DISK PROTECTED?
 6 = DRIVE 2 READY?
 7 = DRIVE 2 VARIABLE LENGTH DISK?
 8 = DRIVE 2 DISK PROTECTED?
 9 = INVOKED BY TERMINAL?
10 = INVOKED BY MODEM?
11 = PARITY ERROR DETECTED (TERMINAL)?
12 = FRAMING ERROR DETECTED (TERMINAL)?
13 = PARITY ERROR DETECTED (MODEM)?
14 = FRAMING ERROR DETECTED?
15 = STANDBY MODE?
16 = BINARY SWITCH IN ENTER POSITION?
17 = PRINTER CHARACTER PRESENT?
```

SGN

Format: SGN(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: The sign of "X".

```
-1 if "X" is negative
 0 if "X" is zero
 1 if "X" is positive
```

Example: A = SGN(3-5) A is equal to -1

SIN

Format: SIN(X)

where: X = A numeric constant, a numeric variable, or a numeric expression which represents the measure of an angle in radians.

Result: The sine of "X".

Example:

A = SIN(.5) A is equal to .47942554

SPC

Format: SPC(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: This function can be used with the PRINT command. It causes "X" number of spaces to be displayed on the terminal device.

The value of "X" must be equal to or greater than zero and equal to or less than 255.

Example:

```
>PRINT 1;SPC(10);2:PRINT 1;"          ";2
 1           2
 1           2
```

SQR

Format: SQR(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: The square root of "X".

Example: A = SQR (9) A is equal to 3

SRCH

SRCH

Formats:

SRCH (<String 1>,<String 2>)
SRCH (<String 1>,<String 2>,<pos>)

where: String 1 = a character string which may be a character constant, variable or expression.

String 2 = a character string, containing the characters searched for in String 1; it may be a character constant, variable or expression.

pos = the character position in String 1 (where the search begins) which may be a numeric variable, constant or expression.

Purpose:

To find the starting position of the first occurrence of specified characters within a character string.

Description:

The result of this function is a numeric value equal to the starting position in String 1 where the character(s) in String 2 begin. String 1 is scanned (from left to right) to find the contents of String 2 within it. If a match is found, the result is equal to the character position in String 1 which contains the character equal to the first character in String 2.

When Comm-Stor performs this function, it evaluates certain relationships among the specified options in a specific order. These evaluations and their results are described below.

1. If the number of characters in String 1 is less than that of String 2, the result will equal zero.

2. If String 1 is a null string, the result will equal zero.

3. If String 2 is a null string, the result will equal the value of "pos". If "pos" is omitted, the result will equal one.

4. If the entire contents of String 1 are searched and the contents of String 2 were not found, the result will equal zero.

5. If "pos" is not specified, or is specified and equal to one, the search will begin from the left-most (first) character position in String 1.

6. If "pos" is specified, its integer value must be greater than zero and less than or equal to 255. The search will begin at that character position in String 1. If "pos" is greater than the length of String 1, the result will equal 0.

Examples:

Command	Result
X=SRCH("ABC","A")	X = 1
X=SRCH("CATABCAT","CAT",3)	X = 6
X=SRCH("ABC","D")	X = 0

Contents of String 1	Contents of String 2	Position	Result
null	A	1	0
null	A1BC	5	0
A1BC	null	5	0
A1BC	null	4	4
ABC	null	*N/S	1
ABC	D	5	0
ABC	D	*N/S	0
ABC	A	*N/S	1
ABC	A	2	0
ABAB	A	*N/S	1
ABAB	A	2	3
ABAB	AB	*N/S	1
ABAB	AB	2	3

*N/S = not specified

STR$

Format: STR$(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: A character string which is the equivalent of the value of "X".

Example:

```
A$ = STR$(2+3)     A$ contains " 5"
```

TAB

Format: TAB(X)

where: X = A numeric constant, a numeric variable, or a numeric expression.

Result: Depending where the output is directed, this function will cause the pointer of the specified device to be repositioned to the character position "X". If the line width of the device is configured to be 0, inaccurate results will occur if "X" is equal to or greater than 256.

If the current position of the pointer is after position "X", a [RETURN] will be issued before the pointer is repositioned to the specified character position.

If the current position of the pointer is equal to "X", the pointer will not be repositioned.

If "X" is greater than the configured line width, the pointer will be repositioned to the value of:

"X"MOD<line width>

which is equal to the remainder obtained from dividing "X" by the line width.

TAN

Format: TAN(X)

where: X = A numeric constant, a numeric variable, or a numeric expression which represents the measure of an angle in radians.

Result: The tangent of "X".

Example:

```
A=TAN(.5)  A is equal to .54630249
```

TIME

Format: TIME(X)

where: X = 0 for the current time
1 for TIMER1
2 for TIMER2

Result: The total number of seconds in the current time or in the TIMERs relative to midnight.

TIME$

Format: TIME$(X)

where: X = 0 for the current time
1 for TIMER1
2 for TIMER2

Result: A character string containing 6 numeric characters which represents the current time or the time in TIMER1 or TIMER2.

Time is returned in 24-hour format.

USF

Format: USF<code>

where: code = A numeric constant or a numeric variable.

Description:

When this function is executed, the system finds the function in the Object Buffer which corresponds to the specified "code". The SYSTEM command must be executed prior to the execution of this function.

VAL

Format: VAL(X$)

where: X$ = A character constant, a character variable, or a character expression.

Result: The numeric value of the contents of "X$".

Example: A=VAL("2") A is equal to 2

APPENDIX A

COMMANDS AND FUNCTIONS

All commands are described in this appendix and are accompanied by their respective formats where:

<xxxx> = the description of the parameter
xxx = the actual (literal) parameter

[xxx] = the parameter is optional
{xxx / yyy} = either xxx or yyy

COMMANDS

APPEND
ASSIGN
AUTO
BYE
CALL
CLEAR
CLEARV
CLOSE
COM
CONT
CREATE
DATA
DEF
DEL
DIM
DIM#
EDIT
END
ERASE
FOR
GOSUB
GOTO
IF END#
IF ERROR#
IF...GOTO
IF...THEN
IF...THEN...ELSE
IMAGE
INPUT
KILL
LET
LINK
LIST
LOAD
MERGE
NEW
NEXT
ON...GOSUB
ON...GOTO
OPEN
OPTION BASE
POKE
POSITION
PRESS
PRINT
PRINT USING
PRINT #
PRINT # USING
RANDOMIZE
READ
READ #
READ # USING
RECSIZE
RELEASE
REM
RENUM
RESTORE
RETURN
REWIND
RTCS
RUN
SAVE
SET
SETIME
STOP
SYSTEM
TRACE
TRUNCATE
UNLOCK
USC
WAIT
WATCH1
WATCH2
WHEN

FUNCTIONS

NUMERIC FUNCTIONS

ABS
ATN
COS
DEC
EXP
FRE
INT
LOG
PEEK
POS
RND
SENSE
SGN
SIN
SQR
TAN
USF

STRING FUNCTIONS

ASC
CHR$
HEX$
LEFT$
LEN
MID$
RIGHT$
SRCH
STR$
VAL

SYSTEM CLOCK FUNCTIONS

TIME
TIME$
CTIME
CTIME$

OUTPUT FUNCTIONS

TAB
FILL
SPC

APPEND (page 7-1)

APPEND#<ref>

Starts sequential access of a disk file after the last character in the file.

```
APPEND#4
APPEND#I
```

ASSIGN (page 7-1)

ASSIGN#<ref>TO[(<drive>)] {*<trk>,<sec> / <file name>}[,<char/rec>][,W][&<sec/buff>[,D]][;#...]

Associates a disk file with a file reference number using the available space in the File Buffer.

```
ASSIGN#4 TO D$
ASSIGN#I TO (2)"TEST",120&2,D
ASSIGN#4 TO A$,50;#7 TO "TEST",30&5
```

AUTO (page 7-4)

AUTO[<line num>][,<inc>]

Automatically supplies line numbers for manually entered BASIC lines.

```
AUTO
AUTO 10
AUTO 32,1
```

BYE (page 7-5)

BYE

Exits BASIC mode and returns the system to Comm-Stor mode.

```
BYE
```

CALL (page 7-5)

CALL[(<DRIVE>)] <file name>

Loads and executes a BASIC program, with the ability to restore and continue the BASIC program which issued the CALL.

```
CALL "SUB3"
CALL A$
CALL (2) LEFT$ (A$,5)
```

CLEAR (page 7-6)

CLEAR[#<ref>]

Deletes all nonconfigured buffers except the BASIC text buffer, or erases the contents of a specific buffer.

```
CLEAR
CLEAR#0
```

CLEARV (page 7-7)

CLEARV

Erases all arrays, variables and constants from User Memory.

```
CLEARV
```

CLOSE (page 7-7)

CLOSE[#<ref>[,T][;#...]]

Deactivates all, or specified, file reference numbers.

```
CLOSE
CLOSE#I
CLOSE#5,T
CLOSE#I;#5,T
```

COM (page 7-9)

COM[#<ref>[&#<ref>[&#...]];]<command>

Allows execution of a Comm-Stor command in BASIC mode.

```
COM".DS"
COM#I;".DD *"
COM#0&#1&#4;".DS"
```

CONT (page 7-10)

CONT

Allows the execution of a program to continue after a stop command or a CNTRL/T has been entered.

```
CONT
```

CREATE (page 7-10)

```
CREATE[(<drive>)]  <file name> [,{ M          }][;[(<drive>)]...]
                                 {<char/file>}
```

Assigns disk space to a specified file name.

```
CREATE"TEST"
CREATE A$,M
CREATE (2) "TEST";A$,100
```

DATA (page 7-11)

```
DATA<val>[,<val>[,...]]
```

Places constants in the internal data table.

```
DATA1
DATA1,"A",C
```

DEF (page 7-12)

```
DEF FN<name>[(<arg>)] = <num exp>
```

Allows the programmer to define a numeric function.

```
DEF FNA (x) = x+1
DEF FNB3(y) = FNA (y)/2
```

DEL (page 7-12)

```
DEL<range>
```

Allows the programmer to delete a range of BASIC lines from User Memory.

```
DEL -10
DEL 10-50
```

DIM (page 7-13)

```
DIM<array name>(<rows>[,<columns>[,...]])[,<array name>...]
```

Specifies the size and name of an array, and reserves space in User Memory for that array.

```
DIM A$(2)
DIM A3 (K)
DIM B (255,2,3),A(5)
```

DIM# (page 7-13)

DIM#<ref>TO {<num array >(<rows>[,<cols>[,...]])
{<char array>(<rows>[,<cols>[,...]]) = <num exp>} [,...][;#...]

Specifies the size and name of a virtual array, and the disk file reference number associated with that array.

```
DIM#6 TO A3 (6)
DIM#5 TO A3 (7),D$ (8)
DIM#7 TO A3 (7);#9 TO A1(5),A$(8) = 40
```

EDIT (page 7-14)

EDIT[<range>]

Allows for changes in a BASIC program line without re-entering the entire line.

```
EDIT 10
EDIT
EDIT -10
EDIT 10-50
EDIT 10-
```

END (page 7-17)

END

Indicates the completed execution of a BASIC program.

```
END
```

ERASE (page 7-17)

ERASE<array list>

Permits erasure of specific arrays from User Memory.

```
ERASE A
ERASE A,A$,F
```

FOR...TO (page 7-17)

FOR<control var> = <init val> TO <final val>[STEP <inc>]

Indicates the number of times that a group of BASIC lines will be executed.

```
FOR I=1 TO 3
FOR I=1 TO Z-3 STEP B
```

GOSUB (page 7-19)

GOSUB<line num>

Transfers program control to a subroutine at the specified line number.

```
GOSUB 2000
```

GOTO (page 7-19)

GOTO<line num>

Transfers program control to the specified line number.

```
GOTO 80
```

IF END# (page 7-19)

IF END#<ref> GOTO { <line num> / END }

Notifies the programmer of an "End of File" condition, and allows for program control.

```
IF END#5 GOTO 200
IF END#I GOTO END
```

IF ERROR# (page 7-20)

IF ERROR#<code> GOTO {<line num> / END}

Notifies the programmer that a specific error has occurred, and allows for program control.

```
IF ERROR#45 GOTO 220
IF ERROR#I GOTO T
```

IF...GOTO (page 7-21)

IF<rel exp> GOTO <line num>

Transfers program control based on the result of the relational expression.

```
IF A$<>B$ GOTO 80
IF NOT A AND B GOTO 75
```

IF...THEN (page 7-21)

IF<rel exp>THEN { <line num> / <command> } [:<command>[:...]]

Allows the programmer to request a specified action, based on a TRUE value of the relational expression.

```
IF A<B THEN A=A+1
IF A$<>B$ THEN PRINT A$:?100
```

IF...THEN...ELSE (page 7-22)

IF<rel exp>THEN { <line num> / <command> } [:<command>[:...]] ELSE <command>[:<command>[:...]]

Allows the programmer to specify particular actions if the relational expression is true, and other actions if that expression is false.

```
IF A<B THEN A=A+1ELSE A=B
IF A$<>B$ THEN PRINT A$:PRINT
200 ELSE IF B$ = C$ THEN PRINT B$
```

IMAGE (page 7-22)

IMAGE<format>

Specifies the format for inputting or outputting data.

```
 50 IMAGE 79#
100 IMAGE 5#;259#
```

INPUT (page 7-23)

INPUT[<char constant>;]<var list>

Assigns constants entered from the terminal to specific variables.

```
INPUT "WHAT IS THE DATE";D$
INPUT A,B,C
```

KILL (page 7-24)

KILL[(<drive>)] <file name>[;[(<drive>)]...]

Deletes the specified disk file from a diskette.

```
KILL A$
KILL A$; (2) "TEST"
KILL (2) "TEST"
```

LET (page 7-24)

[LET]<var> = { <var> | <constant> | <exp> }

Assigns a value to a variable or an array element.

```
LET A=B
C=5
LET A$=LEFT $ (Y$,5)
D=F *20
```

LINK (page 7-25)

LINK[(<drive>)] <file name>

Allows the programmer to enter a new BASIC program while affecting only the free space and BASIC text buffers, and execute it.

```
LINK "PROG 2"
LINK (2) D$
```

LIST (page 7-25)

LIST[#<ref>[&#<ref>[&#...]];][<range>]

Permits the programmer to output a range of BASIC lines presently residing in User Memory.

```
LIST
LIST 10
LIST -10
LIST 10-50
LIST #3; 10-60
```

LOAD (page 7-27)

LOAD[(<drive>)]<file name>[,R][,<range>]

Enters a previously saved disk file into memory.

```
LOAD "sum"
LOAD (2) A$
LOAD "sum", R
LOAD A$,R, 20-900
```

MERGE (page 7-28)

MERGE[(<drive>)]<file name>[,<range>][;[(<drive>)]...]

Adds a disk file text program, previously saved, to the existing program in User Memory.

```
MERGE "sum"
MERGE (2) A$, 20-90
MERGE A$, 10-60; "sum"
```

NEW (page 7-28)

NEW[[(<drive>)]<file name>]

Deletes all buffers in User Memory, except the configured buffers.

```
NEW
NEW SUM
```

NEXT (page 7-29)

NEXT[<control var>]

Denotes the end of that group of BASIC lines starting with a FOR command.

```
NEXT
NEXT I
NEXT I: NEXT J
```

ON...GOSUB (page 7-29)

ON<num exp> GOSUB<line num list>

Uses the rounded whole number result of the specified expression as an index to the list of subroutine line numbers.

```
ON A GOSUB 10,20,30
ON A+B GOSUB 110,120,130,140
```

ON...GOTO (page 7-30)

ON<num exp> GOTO <line num list>

Uses the rounded whole number result of the specified expression as an index to the list of line numbers.

```
ON A GOTO 10,20,30
ON A+B GOTO 110,120,130,140
```

OPEN (page 7-30)

OPEN#<ref> TO { [(<drive>)] { <file name> / *<trk> , <sec> } [,<char/rec>][,W][&<sec/buff>[,D]] / +[&<sec/buff>[,D] } [;#...]

By activating a file reference number, memory in the File Buffer is reserved for inputting and outputting data and for characteristics of the reference number, itself. This command may be used to change the size of the modem buffer.

```
OPEN #4 TO +
OPEN #4 TO *20,15
OPEN #5 TO +&3,D
OPEN #4 TO "TEST",30
OPEN #6 TO +;#5 TO "TEST", 50,W
OPEN #4 TO (2) "TEST" 40&2;#6 TO+
```

OPTION BASE (page 7-32)

OPTION BASE<num exp>

Changes the base of subscripts for all arrays in use.

```
OPTION BASE 1
OPTION BASE A
```

POKE (page 7-33)

POKE<address>,<val>

Enters the specified value into the specified address.

```
POKE A,B
POKE 250,J
POKE J,250
POKE 250,250
```

POSITION (page 7-33)

POSITION#<ref>,<rec>

Positions the first boundary to a record in random access disk files.

```
POSITION #4,2
POSITION #I,J
```

PRESS (page 7-34)

PRESS[(<drive>)]<file name>[,<range>]

Merges a previously saved BASIC text program deleting all REM commands and spaces.

```
PRESS "PROG 1"
PRESS (2) A$
PRESS "PROG 1", 10-68
PRESS A$, 100-520
```

PRINT (page 7-34)

PRINT [<list>[;]]

Displays specified values on the terminal device.

```
PRINT
PRINT A, C+A; A$
PRINT "HELLO",;
PRINT 5
```

PRINT USING (page 7-35)

PRINT USING <image>;[<list>[;]]

Displays specified values on the terminal device according to user specified format.

```
PRINT USING 200;A$
PRINT USING "26#";A$
PRINT USING A$;"HELLO"
```

PRINT # (page 7-36)

PRINT #<ref>[,<rec>][[&#<ref>[,<rec>][&#...]];[<list>[;]]

Output specified values to a device or disk file.

```
PRINT #4
PRINT #1 & #4;A
PRINT #R & #I;A,B;e$
```

PRINT # USING (page 7-36)

PRINT #<ref>[,<rec>][[&#<ref>[,<rec>][&#...]]USING<image>;[<list>[;]]

Output specified values to a terminal device according to a user-specified format.

```
PRINT #5 USING 200;A$
PRINT #0&#2&#4, J USING "26#";A$
PRINT #I, 10 USING R$; A$,B,C;
```

RANDOMIZE (page 7-37)

RANDOMIZE

Generates a new sequence of random numbers.

```
RANDOMIZE
```

READ (page 7-37)

READ<var list>

Assigns values from an internal data table to specified variables or array elements.

```
READ A,B$
READ A3 (7),B$(K), F#
```

READ # (page 7-38)

READ #<ref>[,<rec>];<var list>

Assigns values from a device or disk file to specified variables or array elements.

```
READ #5; A$
READ #I,J;B,C
READ #5,2;B$,A
```

READ # USING (page 7-39)

READ #<ref>[,<rec>] USING<image>;<var list>

Assigns values from a device or disk file to specified variables or array elements according to a user-specified format.

```
READ #5, USING 200;A
READ #I,J USING R$; A,B,C$(K)
READ #1 USING "26#";A$
```

RECSIZE (page 7-40)

RECSIZE#<ref> TO <num exp>

Changes the record size associated with the specified file reference number.

```
RECSIZE #4 TO 80
RECSIZE #I TO A+B
```

RELEASE (page 7-40)

RELEASE[#<ref>[,T][;#...]]

Deactivates a disk file reference number releasing that space in the file buffer for later use.

```
RELEASE #5
RELEASE #5, T;#J
RELEASE #5,T
```

REM (page 7-42)

REM[<comment>]

Allows documentation or remarks to reside in a program, but not be operated on by the system.

```
REM
REM THIS IS A REMARK
```

RENUM (page 7-42)

RENUM[<line num>][,<inc>]

Renumbers the BASIC line numbers residing in User Memory.

```
RENUM
RENUM 3,1
RENUM 20
```

RESTORE (page 7-43)

RESTORE[<line num>]

Permits internal data tables to be reused in whole or in part.

```
RESTORE
RESTORE 59
```

RETURN (page 7-44)

RETURN

Denotes the end of a subroutine, a CALLed program, or an RTCS interruption program, and transfers control to the next line after the command.

```
RETURN
```

REWIND (page 7-44)

REWIND#<ref>

Returns disk file access to sequential access starting at the beginning of the file.

```
REWIND #4
REWIND #I
```

RTCS (page 7-44)

RTCS<num exp>

Enables or disables the Real-Time Control System.

```
RTCS 1
RTCS A1
```

RUN (page 7-45)

RUN[<line num>]

Executes the BASIC program residing in User Memory.

```
RUN
RUN 200
```

SAVE (page 7-45)

SAVE[(<drive>)][<file name>[,T][,<range>]][;[(<drive>)]...]

Saves the current BASIC program in that disk file associated with the indicated file name.

```
SAVE
SAVE (2) F$, T
SAVE "PROG 1"
SAVE (2) F$,T,10-50; "PROG 1"
```

SET (page 7-46)

SET { $! % # } <range> [;{ $! % # } <range>[;...]]

Declares a range of variable names as particular data types.

```
SET $A
SET !A-F
SET %G;#H;%A-F
```

SETIME (page 7-47)

SETIME #<code> TO <char code>

Issues the current time or interrupt times into the Real-Time Control System.

```
SETIME #0 TO A$
SETIME #1 TO "081433"
```

STOP (page 7-48)

STOP

Suspends execution of a BASIC program.

```
STOP
```

SYSTEM (page 7-48)

SYSTEM [(<drive>)] <file name>

Loads a Sykes supplied object program into the object buffer.

```
SYSTEM "ADD"
SYSTEM (2) A$
```

TRACE (page 7-48)

TRACE[#<ref>[&#<ref>[&#...]];]<log exp>

Displays the numbers of the BASIC lines on the specified device as they are executed.

```
TRACE 0
TRACE #3;A1
```

TRUNCATE (page 7-49)

```
TRUNCATE[(<drive>)]<file name>[;[(<drive]...]
```

Deletes any unused space from a specified disk file.

```
TRUNCATE "TEST"
TRUNCATE "TEST";(2)A$
TRUNCATE (2) A$
```

UNLOCK (page 7-50)

```
UNLOCK #<ref>
```

Returns disk file access to sequential access starting at the current position in the file.

```
UNLOCK #4
UNLOCK #I
```

USC (page 7-50)

```
USC<code>
```

Executes commands residing in the object buffer.

```
USC 20
USC A
```

WAIT (page 7-50)

```
WAIT<address>,<val 1> , <val 2>
```

Delays execution of the current program until the specified location contains the necessary operative value.

```
WAIT 650, 5
WAIT A,B,C
```

WATCH (page 7-51)

```
WATCH{1}<address><val 1>[,<val 2>]
     {2}
```

Prepares those hardware registers to be used in the Real-Time Control System.

```
WATCH 1 650, 5
WATCH 2 A,B,C
```

WHEN (page 7-52)

```
WHEN #<ref> { GOSUB } <line num> [ {&<priority>}[,E] ]
            { GOTO  }             {&E             }
            { END   }
```

Controls the actions to be taken when the specified interrupt is detected.

```
WHEN #4 GOTO 500
WHEN #7 GOSUB 5000, 8
WHEN #1 GOTO 100,E
WHEN #6 END
```

This page intentionally blank.

APPENDIX B

ALLOCATING USER MEMORY

CONTENTS

The Comm-Stor User Memory Space can contain up to eight different buffer areas. Since these buffers interrelate, it is important that the advanced programmer understands how these buffers are used. The eight buffers are pictured in Figure #B-1 and all may be used in BASIC mode. Only two may be used in Comm-Stor mode and are denoted by asterisks.

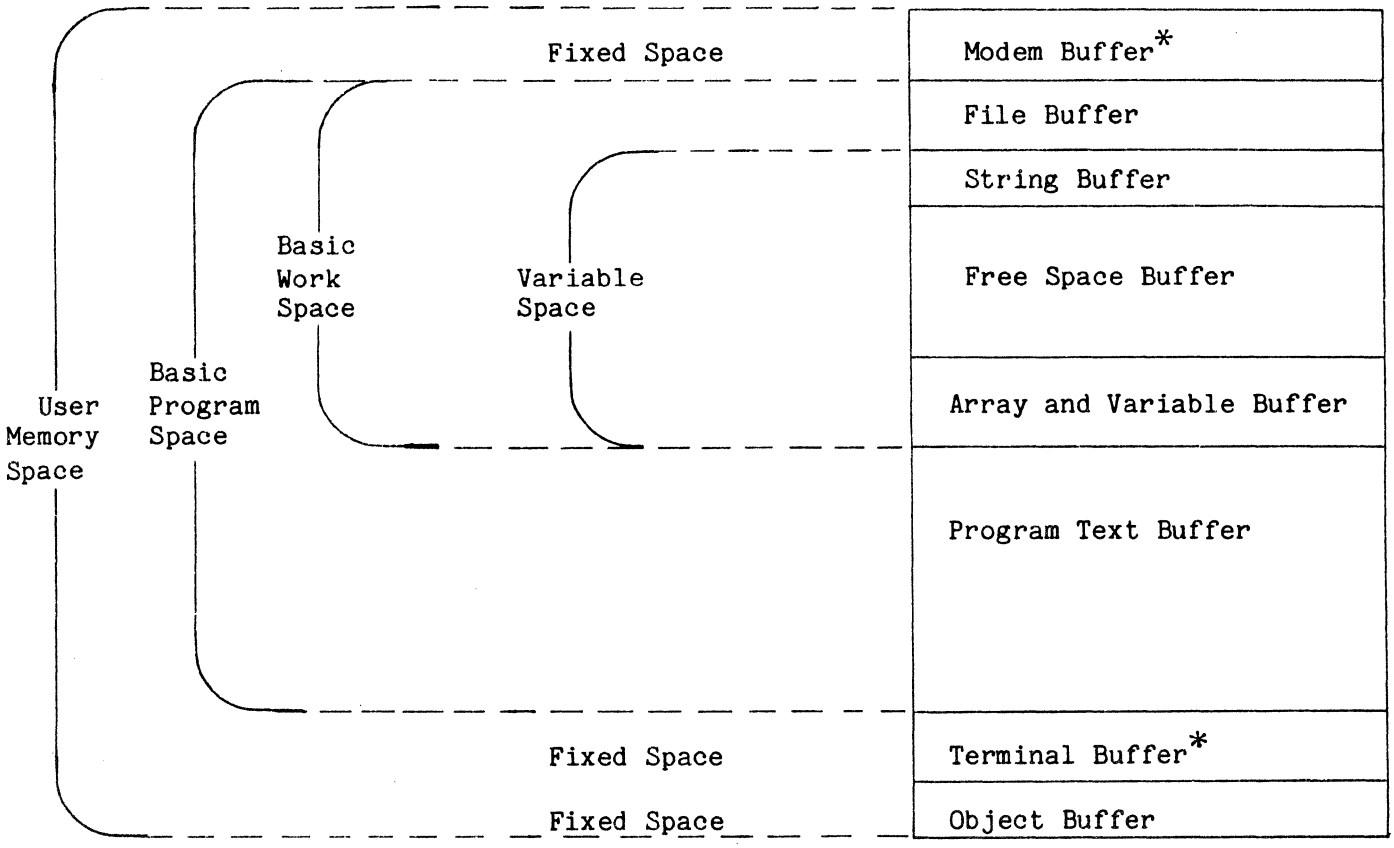


FIGURE #B-1

1. THE MODEM BUFFER

The Modem Buffer temporarily holds incoming data from the device connected to the modem port. This buffer may be used in both the Comm-Stor and BASIC modes.

The size of this buffer is specified at configuration time; a maximum of 32K bytes is available. When the Comm-Stor unit is powered-up, the amount of memory specified will be allocated for this buffer. Its size, however, can be changed by the programmer if desired. This is done by using the ".AL" command in Comm-Stor mode. For example, the command

```
.AL M/5235
```

will allocate 5235 bytes of User Memory for the Modem Buffer. When this command is executed, the contents of the BASIC Program Space are destroyed becoming part of the Free Space Buffer which will be the only buffer existing in the BASIC Program Space.

In BASIC mode, the size of the Modem Buffer may be changed by using the OPEN command. For example, the command

```
OPEN #2 TO + & 35
```

will allocate 4480 bytes (35 sectors x 128 bytes = 4480 bytes) of User Memory for the Modem Buffer. When this command is executed, the contents of the BASIC Work Space are destroyed becoming part of the Free Space Buffer. Only the Free Space Buffer will exist in the BASIC Work Space. Therefore, OPENing the Modem Buffer should be the first command in the BASIC program.

2. THE TERMINAL BUFFER

The Terminal Buffer temporarily holds incoming data from the device connected to the terminal port. This buffer may be used in both the Comm-Stor and BASIC modes. The size of this buffer is specified at configuration time and cannot be changed by any Comm-Stor or BASIC command. A maximum of 32K bytes is available for terminal buffering.

3. THE OBJECT BUFFER

The Object Buffer holds BASIC object code which is loaded into the User Memory Space by using the SYSTEM command. The size of this buffer is specified at configuration time and cannot be changed by any BASIC command. A maximum of 16K bytes is available for object buffering.

4. THE FILE BUFFER

The File Buffer temporarily holds data transferred to and from disk files. The size of this buffer is determined by the number of sectors used in OPEN commands for disk file reference numbers (see page 5-3). Once a character variable is used in BASIC mode, the size of the buffer may no longer be changed. Therefore, the total size needed for all disk file operations should be reserved before referencing any character variables.

5. THE STRING BUFFER

The String Buffer holds any character strings used in BASIC mode. When a character constant is specified in a command, it is temporarily held in this buffer and then deleted once the command has been executed. Once a character variable is used in a command, however, its contents are entered into this buffer and will not be deleted unless specified by another command.

6. THE PROGRAM TEXT BUFFER

The Program Text Buffer holds all BASIC program lines used in BASIC mode.

7. THE ARRAY AND VARIABLE BUFFER

The arrays and variables used in BASIC mode are stored in this buffer. When character variables and character arrays are used, their names are placed in this buffer accompanied by an address to locate their contents in the String Buffer. Additionally, any functions created by a DEF command are stored in this buffer.

8. THE FREE SPACE BUFFER

The Free Space Buffer is the amount of unused space which exists in the User Memory Space. The numeric function FRE is used by the programmer to determine the size of this buffer.

9. ALLOCATING BUFFERS

As described previously, the Modem, Terminal and Object Buffer sizes are specified at configuration time. When the Comm-Stor unit is powered-up, these buffers are automatically allocated in the User Memory Space.

For example, assume that the total size of the User Memory Space is 16K (or 16,000 bytes). Suppose too, the Modem Buffer was configured to be 1280 bytes, the Terminal Buffer to be 526 bytes and the Object Buffer to be 2048. After powering up Comm-Stor, the User Memory Space can be illustrated in Figure #B-2.

Entering the .BA command at this time places the system into the BASIC mode and the Direct mode. The Free Space Buffer now contains 12,146 bytes (becauses 16,000 - 1,280 - 256 - 2,048 = 12,146). LOAD a BASIC program in packed format containing 370 characters. Since the program is in packed format, its size is rounded up to the nearest sector for loading purposes. Therefore, there must be enough room in the User Memory Space to accommodate the required number of bytes. Once the pro-

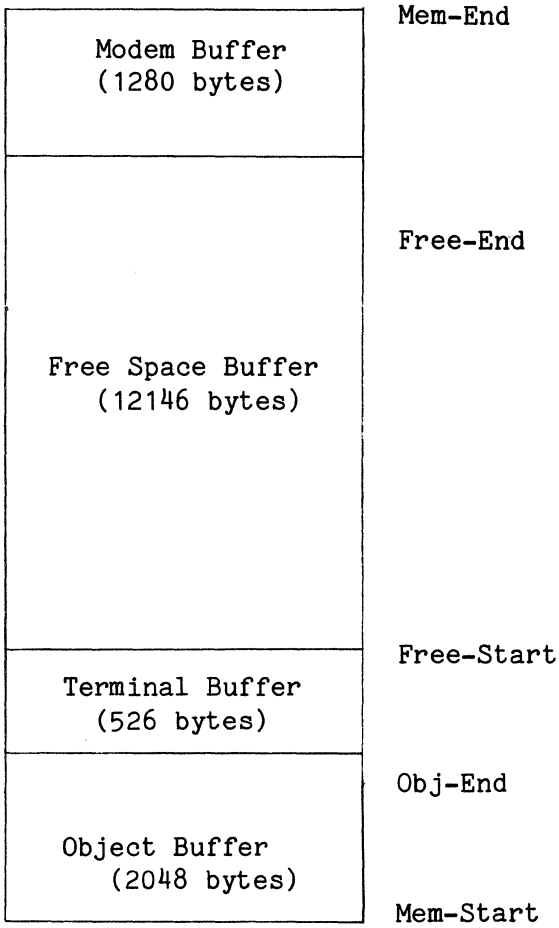


FIGURE #B-2

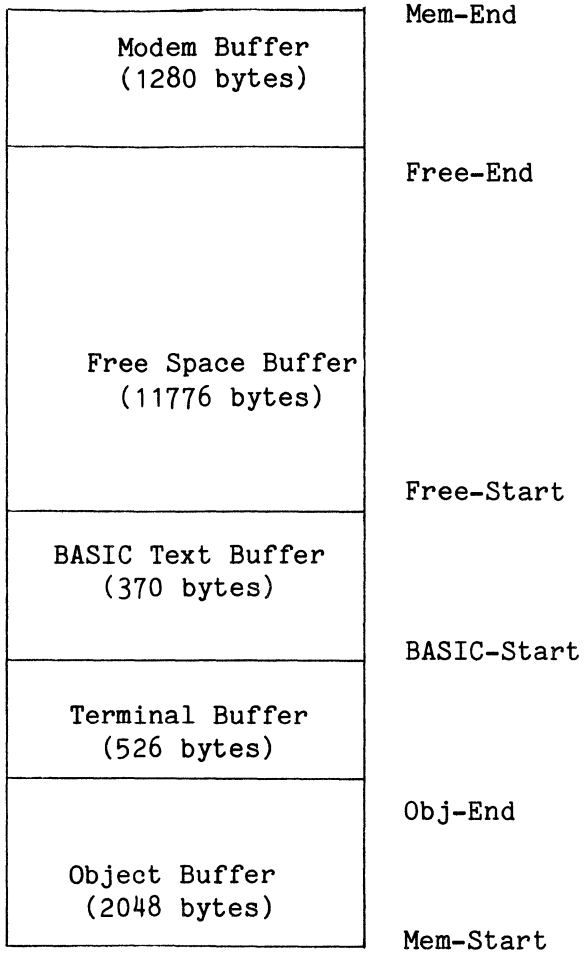


FIGURE #B-3

gram is loaded into the Program Text Buffer, the buffer is reduced to the actual character count of the program. For example, the Program Text Buffer is first allocated 384 bytes, or 3 sectors (370 characters + 128 characters = 2.89 sectors). The program is then loaded into this buffer which is then reduced to 370 bytes and the User Memory Space will now look like Figure #B-3. The Free Space Buffer now contains 11,776 bytes (because 12,146 - 384 = 11,762).

The required number of bytes used by a BASIC program in text format can be determined by loading the program, saving it in packed format, and then looking at the byte count (in a long Directory display, ".DS L").

Suppose we now want to change the size of the Modem Buffer. As we have only loaded our BASIC program and did not allocate any other nonconfigured buffers, now would be the best time to do this.

Suppose the first line of the BASIC program is

```
OPEN #2 TO + & 9
```

and we execute the program. Since this command changed the size of the Modem Buffer to 9 sectors, the User Memory Space now looks like Figure #B-4.

Now, in order to transfer information to and from a disk file, the next command in the BASIC program should be an OPEN

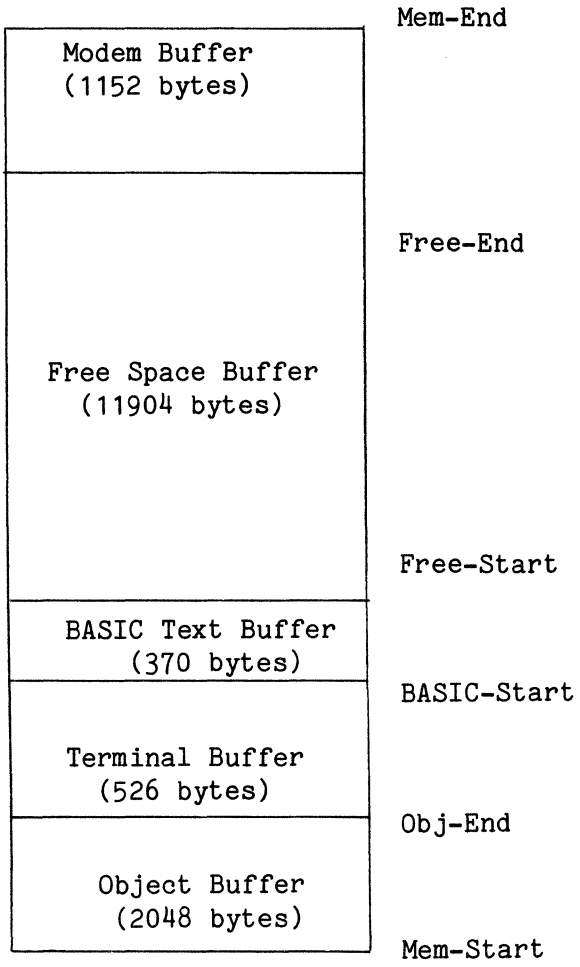


FIGURE #B-4

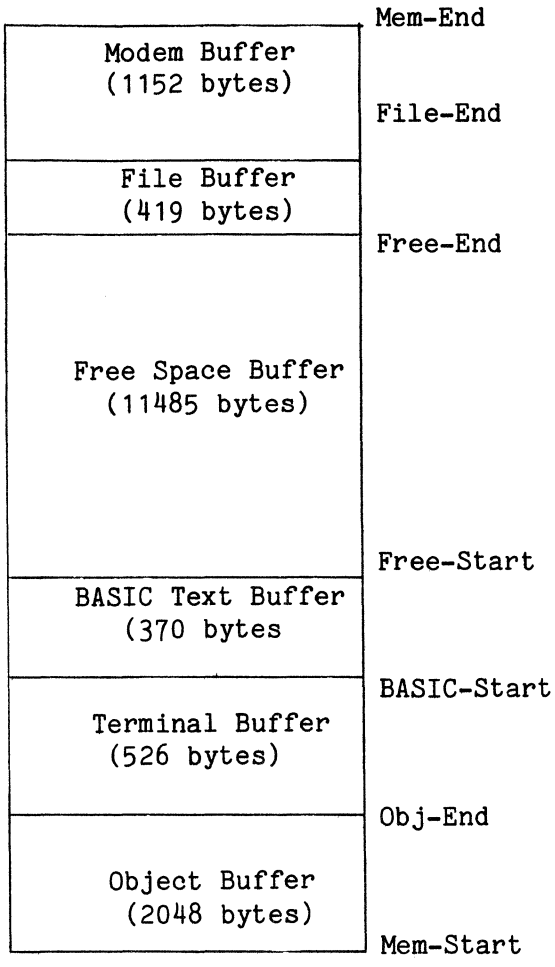


FIGURE #B-5

command using a disk file reference number such as:

OPEN #4 TO + & 3

This command allocates 3 sectors of file space for temporarily holding data plus 35 bytes for holding the different parameters associated with disk file reference number 4. The User Memory Space now appears as in Figure #B-5. The space needed for the File Buffer was taken from the Free Space Buffer.

Now, suppose the BASIC program performs some numeric calculations and OPENs another disk file reference number reserving 2 more sectors. The User Memory Space could then be shown as in Figure #B-6.

The size of the File Buffer was increased by 291 bytes (2 sectors x 128 bytes +35 bytes = 291 bytes) which were taken from the Free Space Buffer. Additionally, the numeric calculations used some variables and, consequently, the Array and Variable Buffer was allocated (for this example, 16 bytes). These bytes were also taken from the Free Space Buffer.

Now, suppose the next line in the BASIC program is

A$ = "TEST"

This command stores the characters "TEST" in the String Buffer. The name A$ and an address which points to the above

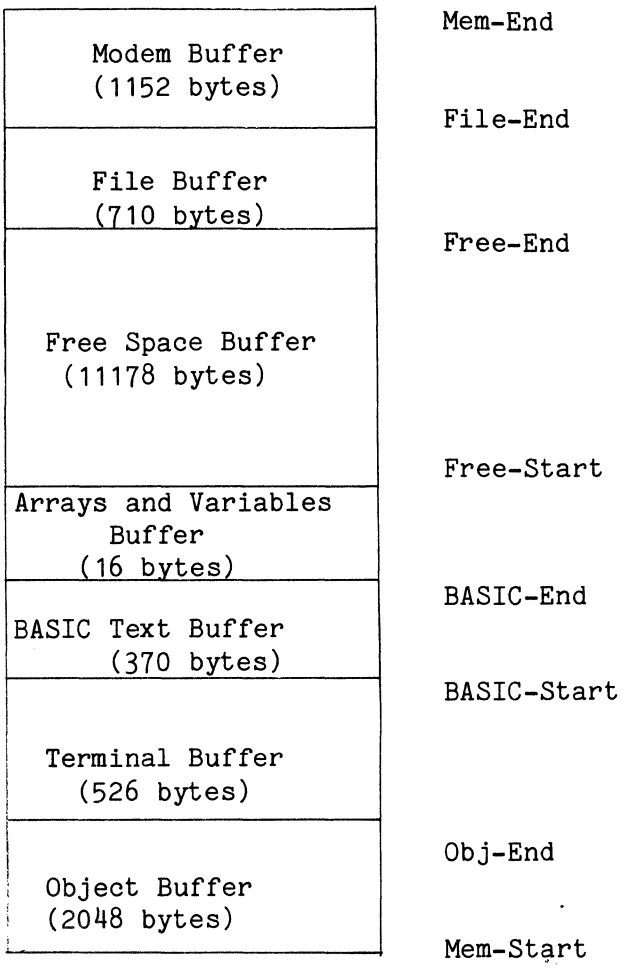


FIGURE #B-6

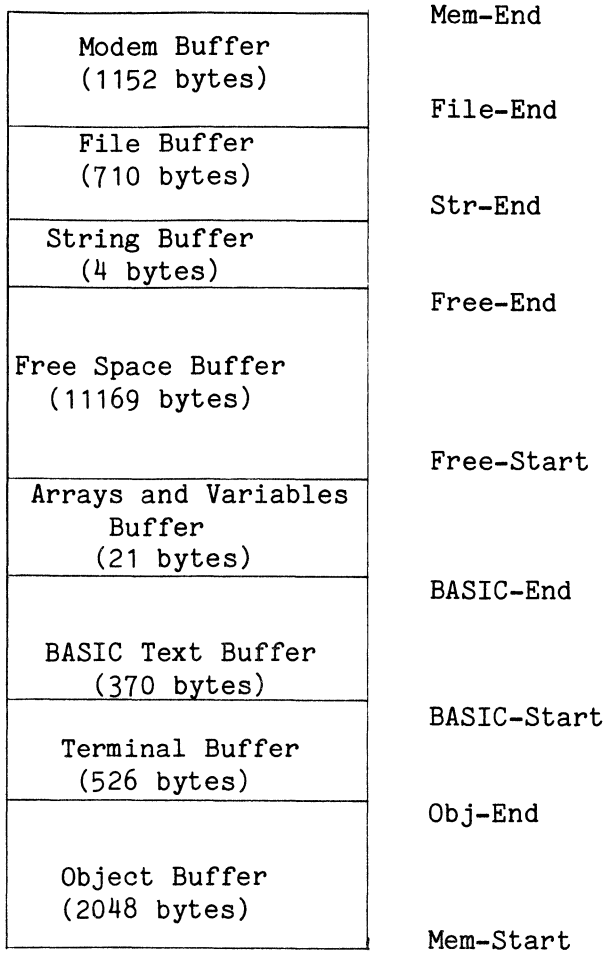


FIGURE #B-7

characters are stored in the Array and Variable Buffer. By allocating space in the String Buffer, the size of the File Buffer is fixed. The User Memory Space now is shown in Figure #B-7. Five bytes were needed to store the name and address of A$ in the Array and Variable Buffer, and 4 bytes were needed to store the four were needed to store the four characters "TEST" in the String Buffer. As you can see from Figure #B-7, a disk file reference number can no longer be OPENed because the extra File Buffer space would have to be obtained from the String Buffer which is not allowed.

10. COMMANDS AFFECTING THE USER MEMORY SPACE

The commands:

CLEAR
CLEARV
LINK/CALL
LOAD
MERGE
NEW

are discussed in this chapter because the requirements for buffering in the User Memory Space play a major role in the execution of these commands. Prior to the description of each command, the User Memory Space will appear as in Figure #B-8.

A. CLEAR

When a CLEAR command is executed, the contents of the BASIC Work Space are de-

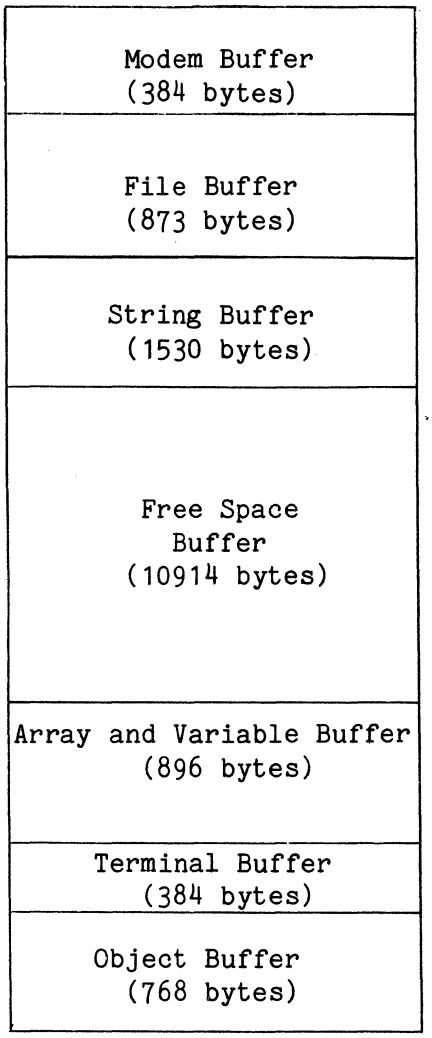


FIGURE #B-8

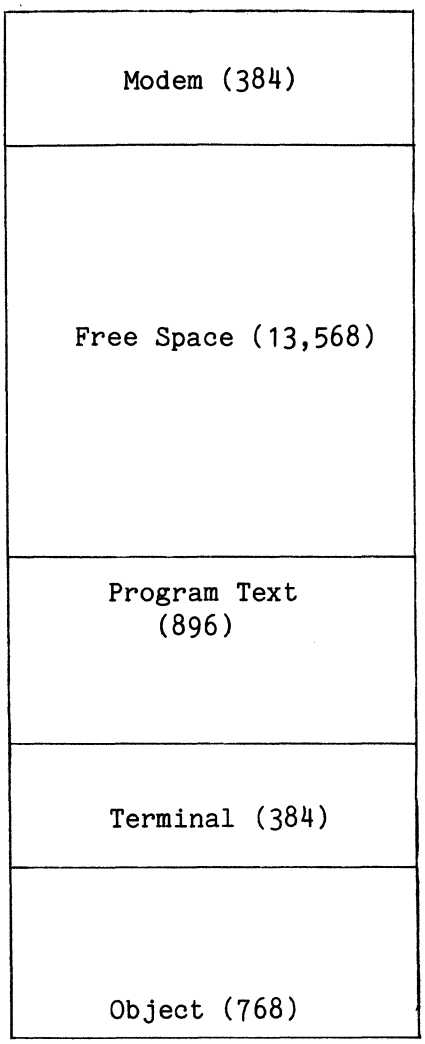


FIGURE #B-9

leted and these bytes are returned to the Free Space Buffer. Additionally, if any IF END # or IF ERROR # commands were executed prior to this command, they will be deleted from Comm-Stor. Also, if an OPTION BASE, a SET, or RTCS INTERRUPT command were executed prior to the CLEAR, they will be returned back to their original default values. After executing a CLEAR command, the User Memory Space would look like Figure #B-9.

B. CLEARV

When a CLEARV command is executed, the contents of the Variable Space are deleted and these bytes are returned to the Free Space Buffer. Everything else will remain the same except that any programmer-defined functions and the precisions specified in a SET command will be deleted. If we were to execute a CLEARV command, the User Memory Space would look like Figure #B-10.

C. LINK/CALL

The LINK command enables the programmer to enter a new BASIC program into the User Memory Space affecting only the Free Space Buffer and the Program Text Buffer. When a LINK command is executed, Comm-Stor determines if the number of bytes in the Free Space Buffer plus the number of bytes in the Program Text Buffer is large enough to accommodate the new program. If it is, the number of bytes in the new program is rounded up to the nearest sector. Comm-

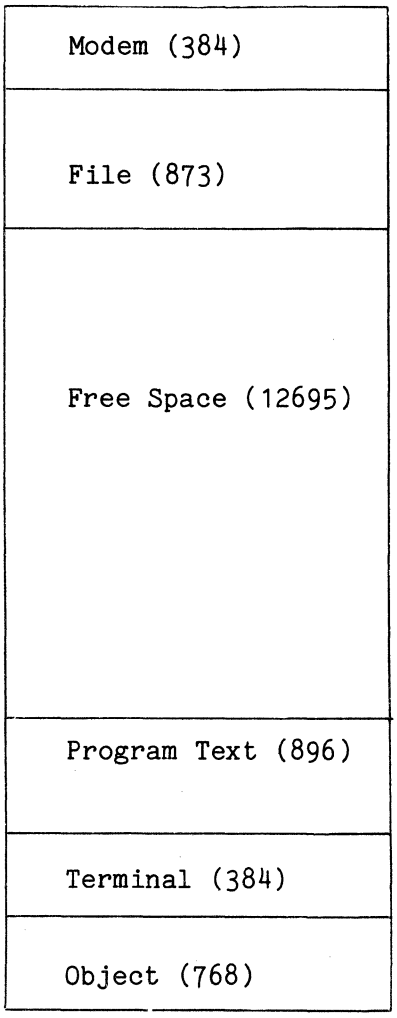


FIGURE #B-10

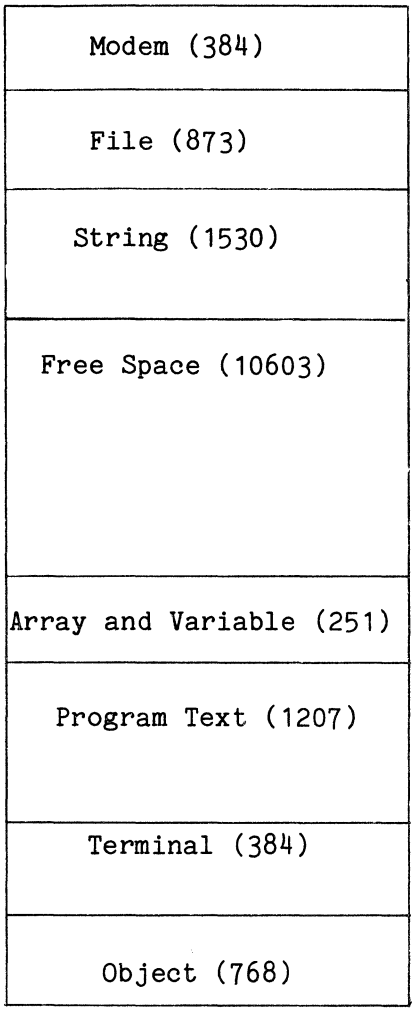


FIGURE #B-11

Stor then determines if the Program Text Buffer is large enough to contain the required number of bytes. If it is not, enough bytes are transferred from the Free Space Buffer into the Program Text Buffer. The new program is then entered into the Program Text Buffer. Any unused bytes will be transferred back into the Free Space Buffer and the new program is automatically executed. A LINK causes DEF and IF ERROR # commands to be CLEARed. CALL works similarly in allocating Buffer Space.

For example, consider a program saved in packed format which contains 1207 bytes. If we executed the LINK command for this program, the User Memory Space would be like Figure #B-11.

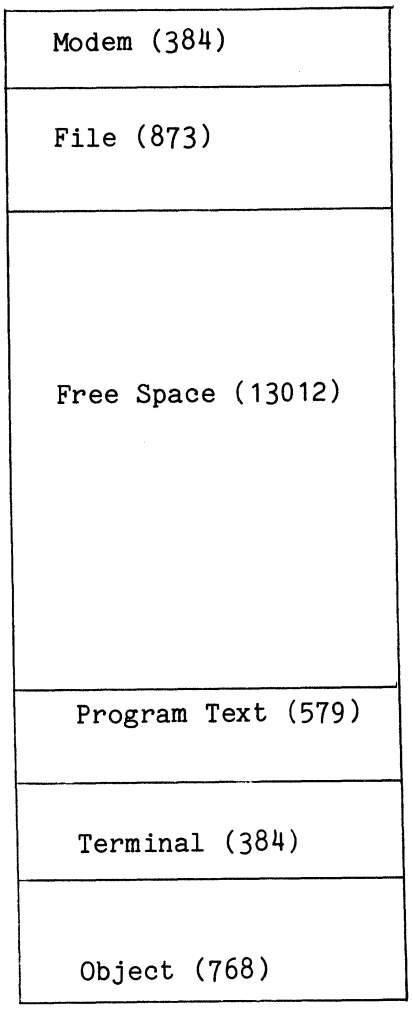


FIGURE #B-12

D. LOAD

Loading a program in Direct mode is previously described in this appendix (page B-3). A LOAD and RUN command may also be executed within a BASIC program. When this is done, however, the operation is different from that previously described. Suppose a BASIC program is executing and the command

LOAD "PROGRAM 1",R

is in the program. Assume "PROGRAM 1" is a program in packed format and contains 579 bytes. If the above command is executed from within the current BASIC program, Comm-Stor will issue a CLEARV before entering the new program into memory (and

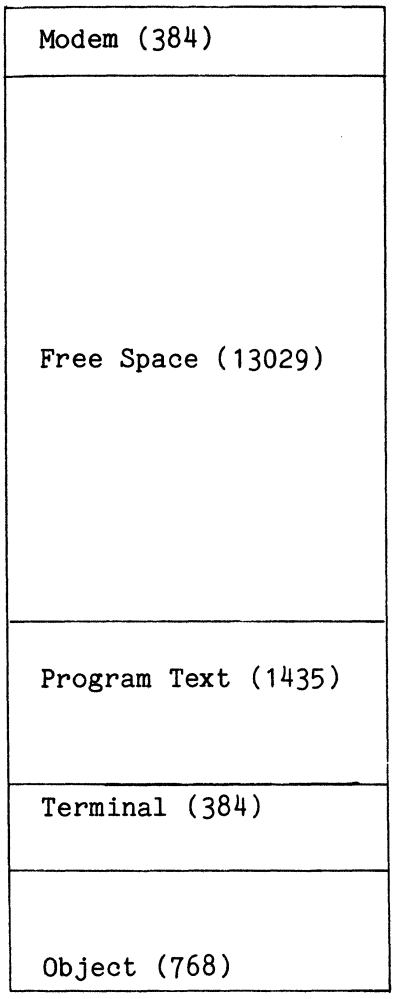


FIGURE #B-13

executing it). Therefore, if the above command was executed, the User Memory Space would appear as in Figure #B-12.

E. MERGE

The MERGE command allows the programmer to add a BASIC program saved in text format to the current contents of the Program Text Buffer. Suppose we wanted to MERGE a program which contained 596 ASCII characters. Suppose too, that the number of actual bytes (in packed format) used by the program is 539. When a MERGE command is executed, the system issues a CLEAR before the program is added to the Program Text Buffer. If we executed a MERGE with the above program, the User Memory Space would appear as in Figure #B-13.

F. NEW

A NEW command is used to tell the system to delete the contents of the BASIC program space and to return these bytes to the Free Space Buffer. Therefore, if we issued a NEW command, the User Memory Space would look like Figure #B-14.

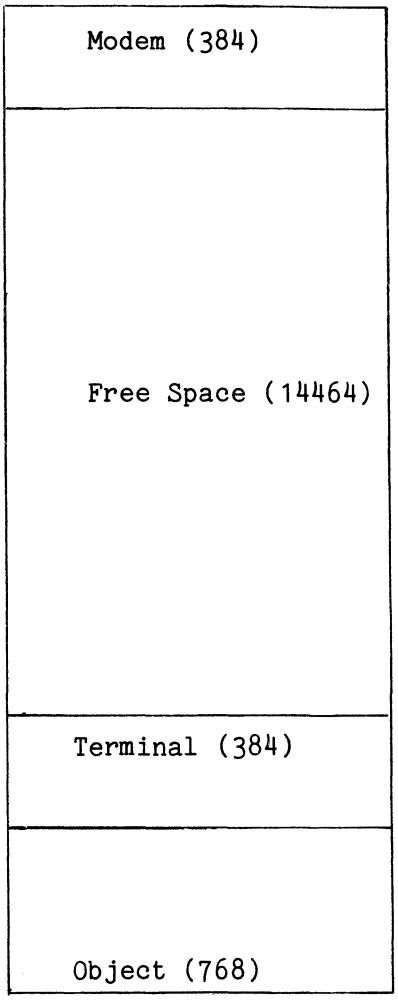


FIGURE #B-14

This page intentionally blank.

APPENDIX C

ASCII/EBCDIC CODE CHART

CHARACTER CODE TABLE:

DECIMAL/OCTAL/HEXADECIMAL TO EBCDIC/ASCII

Decimal	Octal	Hexadecimal	EBCDIC	ASCII	Decimal	Octal	Hexadecimal	EBCDIC	ASCII
0	000	00	NUL	NUL	33	041	21	SOS	!
1	001	01	SOH	SOH	34	042	22	FS	"
2	002	02	STX	STX	35	043	23		#
3	003	03	ETX	ETX	36	044	24	BYP	$
4	004	04	PF	EOT	37	045	25	LF	%
5	005	05	HT	ENQ	38	046	26	ETB	&
6	006	06	LC	ACK	39	047	27	ESC	'
7	007	07	DEL	BEL	40	050	28		(
8	010	08		BS	41	051	29		)
9	011	09		HT	42	052	2A	SM	*
10	012	0A	SMM	LF	43	053	2B	CU2	+
11	013	0B	BT	VT	44	054	2C		,
12	014	0C	FF	FF	45	055	2D	ENQ	-
13	015	0D	CR	CR	46	056	2E	ACK	.
14	016	0E	SO	SO	47	057	2F	BEL	/
15	017	0F	SI	SI	48	060	30		0
16	020	10	DLE	DLE	49	061	31		1
17	021	11	DC1	DC1	50	062	32	SYN	2
18	022	12	DC2	DC2	51	063	33		3
19	023	13	TM	DC3	52	064	34	PN	4
20	024	14	RES	DC4	53	065	35	RS	5
21	025	15	NL	NAK	54	066	36	UC	6
22	026	16	BS	SYN	55	067	37	EOT	7
23	027	17	IL	ETB	56	070	38		8
24	030	18	CAN	CAN	57	071	39		9
25	031	19	EM	EM	58	072	3A		:
26	032	1A	CC	SUB	59	073	3B	CU3	;
27	033	1B	CU1	ESC	60	074	3C	DC4	<
28	034	1C	IFS	FS	61	075	3D	NAK	=
29	035	1D	IGS	GS	62	076	3E		>
30	036	1E	IRS	RS	63	077	3F	SUB	?
31	037	1F	IUS	US	64	100	40	SP	@
32	040	20	DS	SP	65	101	41		A

ASCII/EBCDIC CODE CHART (cont.)

Decimal	Octal	Hexadecimal	EBCDIC	ASCII
66	102	42		B
67	103	43		C
68	104	44		D
69	105	45		E
70	106	46		F
71	107	47		G
72	110	48		H
73	111	49		I
74	112	4A	¢	J
75	113	4B	.	K
76	114	4C	<	L
77	115	4D	(	M
78	116	4E	+	N
79	117	4F	!	O
80	120	50	&	P
81	121	51		Q
82	122	52		R
83	123	53		S
84	124	54		T
85	125	55		U
86	126	56		V
87	127	57		W
88	130	58		X
89	131	59		Y
90	132	5A	!	Z
91	133	5B	$	[
92	134	5C	*	\
93	135	5D	)	]
94	136	5E	;	^
95	137	5F	¬	-
96	140	60	-	`
97	141	61	/	a
98	142	62		b
99	143	63		c
100	144	64		d
101	145	65		e
102	146	66		f
103	147	67		g

Decimal	Octal	Hexadecimal	EBCDIC	ASCII
104	150	68		h
105	151	69		i
106	152	6A	¦	j
107	153	6B	,	k
108	154	6C	%	l
109	155	6D	_	m
110	156	6E	>	n
111	157	6F	?	o
112	160	70		p
113	161	71		q
114	162	72		r
115	163	73		s
116	164	74		t
117	165	75		u
118	166	76		v
119	167	77		w
120	170	78		x
121	171	79	`	y
122	172	7A	:	z
123	173	7B	#	{
124	174	7C	@	:
125	175	7D	'	}
126	176	7E	=	~
127	177	7F	"	DEL
128	200	80		
129	201	81	a	
130	202	82	b	
131	203	83	c	
132	204	84	d	
133	205	85	e	
134	206	86	f	
135	207	87	g	
136	210	88	h	
137	211	89	i	
138	212	8A		
139	213	8B		
140	214	8C		
141	215	8D		

ASCII/EBCDIC CODE CHART (cont.)

Decimal	Octal	Hexadecimal	EBCDIC
142	216	8E	
143	217	8F	
144	220	90	
145	221	91	j
146	222	92	k
147	223	93	l
148	224	94	m
149	225	95	n
150	226	96	o
151	227	97	p
152	230	98	q
153	231	99	r
154	232	9A	
155	233	9B	
156	234	9C	
157	235	9D	
158	236	9E	
159	237	9F	
160	240	A0	
161	241	A1	~
162	242	A2	s
163	243	A3	t
164	244	A4	u
165	245	A5	v
166	246	A6	w
167	247	A7	x
168	248	A8	y
169	251	A9	z
170	252	AA	
171	253	AB	
172	254	AC	
173	255	AD	
174	256	AE	
175	257	AF	
176	260	B0	
177	261	B1	
178	262	B2	
179	263	B3	

Decimal	Octal	Hexadecimal	EBCDIC
180	264	B4	
181	265	B5	
182	266	B6	
183	267	B7	
184	270	B8	
185	271	B9	
186	272	BA	
187	273	BB	
188	274	BC	
189	275	BD	
190	276	BE	
191	277	BF	
192	300	C0	{
193	301	C1	A
194	302	C2	B
195	303	C3	C
196	304	C4	D
197	305	C5	E
198	306	C6	F
199	307	C7	G
200	310	C8	H
201	311	C9	I
202	312	CA	
203	313	CB	
204	314	CC	
205	315	CD	
206	316	CE	
207	317	CF	
208	320	D0	}
209	321	D1	J
210	322	D2	K
211	323	D3	L
212	324	D4	M
213	325	D5	N
214	326	D6	O
215	327	D7	P
216	330	D8	Q
217	331	D9	R

ASCII/EBCDIC CODE CHART (cont.)

Deci-mal	Octal	Hexa-deci-mal	EBCDIC
218	332	DA	
219	333	DB	
220	334	DC	
221	335	DD	
222	336	DE	
223	336	DF	
224	340	E0	\
225	341	E1	
226	342	E2	S
227	343	E3	T
228	344	E4	U
229	345	E5	V
230	346	E6	W
231	347	E7	X
232	350	E8	Y
233	351	E9	Z
234	352	EA	
235	353	EB	
236	354	EC	↵
237	355	ED	
238	356	EE	
239	357	EF	
240	360	F0	0
241	361	F1	1
242	362	F2	2
243	363	F3	3
244	364	F4	4
245	365	F5	5
246	366	F6	6
247	367	F7	7
248	370	F8	8
249	371	F9	9
250	372	FA	!
251	373	FB	
252	374	FC	
253	375	FD	
254	376	FE	
255	377	FF	

APPENDIX D

RESERVED WORDS

Words which have a special meaning to the BASIC Interpreter are called RESERVED WORDS. They command the Comm-Stor IV unit to perform a specific operation.

Since these words are used as commands, they must not be used wtihin a variable name (page 2-4) or within a line. For example, the statement:

IF N>NO THEN...

is unacceptable (the message ERROR-SN is displayed) because BASIC ignores blanks; the characters NO T are interpreted as the reserved word NOT. The statement should be corrected to read:

IF NO<N THEN...

A list of the reserved words is provided below. Comm-Stor IV will act on the first reserved word encountered in a BASIC LINE.

ABS	END	NOT	SETIME
AND	ERASE	ON	SGN
APPEND	ERROR	OPEN	SIN
ASC	EXP	OPTION	SPC
ASSIGN	FILL	OR	SQR
ATN	FN	PEEK	SRCH
AUTO	FOR	POKE	STEP
BASE	FRE	POS	STOP
BYE	GOSUB	POSITION	STR$
CALL	GOTO	PRESS	SYSTEM
CHR$	HEX$	PRINT	TAB
CLEAR	IF	RANDOMIZE	TAN
CLEARV	IMAGE	READ	THEN
CLOSE	INPUT	RECSIZE	TIME
COM	INT	RELEASE	TIME$
CONT	KILL	REM	TO
COS	LEFT$	RENUM	TRACE
CREATE	LEN	RESTORE	TRUNCATE
CTIME	LET	RETURN	UNLOCK
CTIME$	LINK	REWIND	USC
DATA	LIST	RIGHT$	USF
DEC	LOAD	RND	USING
DEF	LOG	RTCS	VAL
DEL	MID$	RUN	WAIT
DIM	MERGE	SAVE	WATCH1
DIM#	NEW	SENSE	WATCH2
EDIT	NEXT	SET	WHEN
ELSE			

Note: An "ERROR - SN" will occur if any of the above words appear within a variable name or BASIC line.

This page intentionally blank.

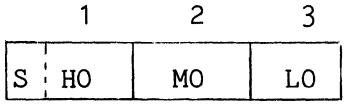
APPENDIX E

INTERNAL FORMATS

Sykes BASIC uses two different types of formats for storing numeric values. They are: INTEGER and FLOATING POINT formats.

1. Integer Format

When a number is specified as an integer, its value is stored in the system using 3 bytes. These bytes, from left to right, are called the HIGH ORDER, MIDDLE ORDER, and LOW ORDER bytes. Each byte contains 8 bits. The first bit of the high order byte is used to contain the sign of the value. Integer format may be illustrated as:



The range of integer values lies between -8388608 and +8388608.

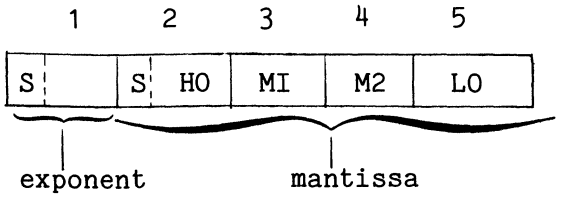
2. Floating Point Formats

When the system stores a value in floating point format, it separates the value into an exponent and a mantissa; and each of these contains a sign. The range of floating point values lies between 10^{-37} and 10^{37}. There are two types of floating point formats: SINGLE PRECISION and EXTENDED PRECISION.

Single Precision Format

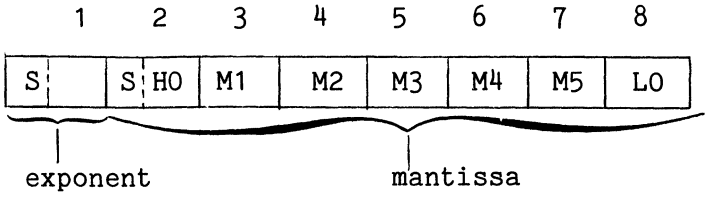
The single precision format is used to store all numeric values unless they are specified otherwise, and provides economical storage capabilities.

This format can be illustrated as:



Extended Precision Format

When a value is declared as an extended precision value, its accuracy is greater than if it was a single precision value. This is because more memory space is used to represent extended precision values; this can be illustrated as:



3. Character Format

Character strings are stored internally as one byte per character. One string can contain up to 255 characters and the internal format of the string depends on the number of characters in that particular string.

This page intentionally blank.

APPENDIX F

Comm-Stor USER DISKETTE FORMATS

User diskettes are created by using the program which resides on the Configuration diskette. The procedure for using this program is described in the Comm-Stor IV Configuration Manual (Sykes Publication No. 9990B0259). The Configuration program creates the Directory (initialized to zero) and writes the Diskette Header in accordance with the parameters specified by the user. The general format of a User diskette is as follows (see Figure #F-1):

Track	Sector	Meaning
0	1-26	IBM Index Track (unused)
1	1	Diskette Header
1	2	Reserved
1	3	Start of Directory
T_1	1	Unused
T_2	1	Start of Library

1. IBM Index Track (Track 0)

The Comm-Stor IV system uses IBM 3741 compatible diskettes. The "Data Set Labels" which reside on the track are prerecorded by IBM and are not affected by

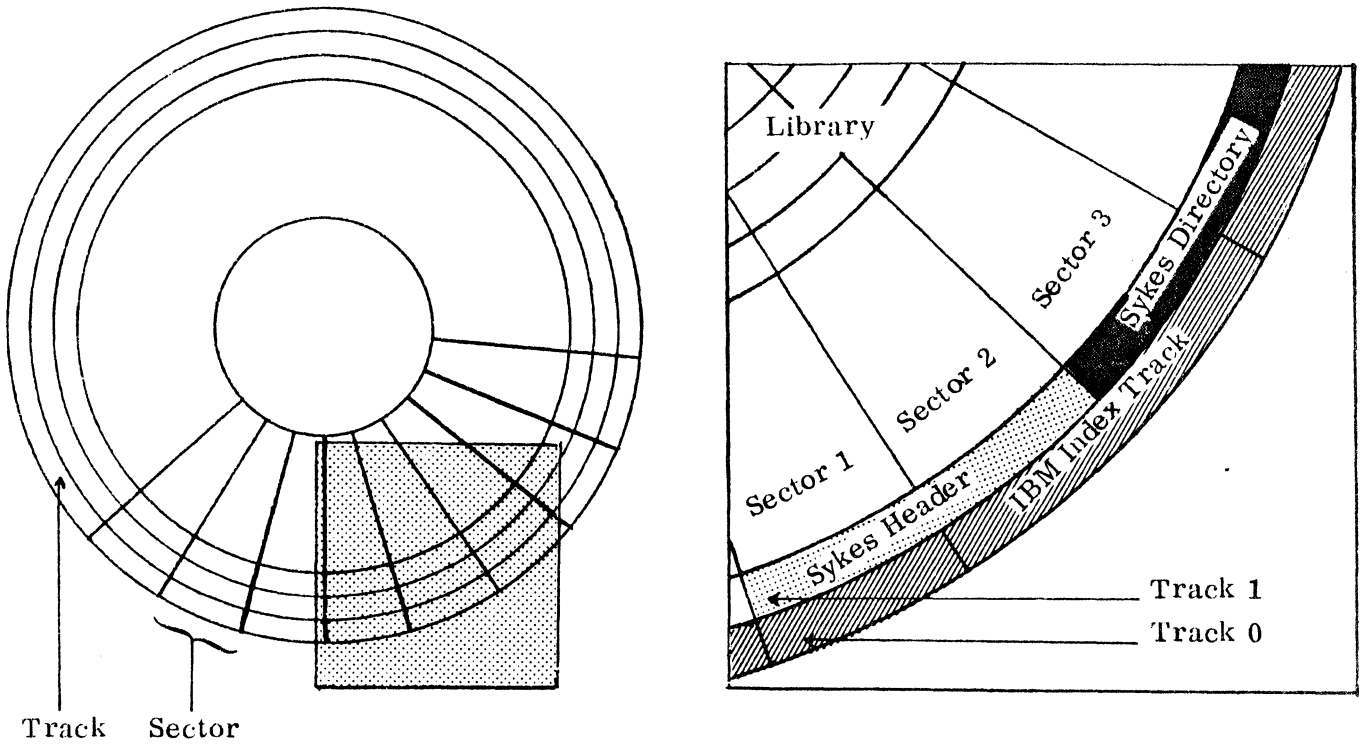


FIGURE #F-1

the Comm-Stor system in any way. (The user may freely READ and PRINT on this track without affecting Comm-Stor operations.)

2. Diskette Header (Track 1, Sector 1)

The Diskette Header contains various parameters which define the characteristics of the files which will reside on the diskette.

The format of the Diskette Header is as follows:

Byte	Use
1-3	ASCII characters "USR"
4	Internal use
5-6	Number of active files in Library
7	Miscellaneous parameters as follows:

Bit Position

7 (MSB)	1=Variable Length files
6	1=Ignore CRC errors
5	1=Diskette is Write Protected
4	1=Double sided diskettes
3	1=Use IBM spare tracks
0-2	Unused

8-9	Unused by Comm-Stor; start of scratch pad
10-11	Total number of Library slots
12-13	If fixed length = Maximum size per Library slot If variable length = Hex. value "0F27" (9999_{10})
14	Maximum lines per message
15	Maximum characters per line
16	Maximum characters per file name
17	Maximum characters per extension
18	Number of characters per Directory Entry
19	Number of Directory Entries per Directory Block
20-21	If fixed length = Unused If variable length = Average file size
22-128	Reserved

NOTES:

1. All bytes contain binary values except 1,2,3, and 7. The byte pairs (5-6), (10-11), and (12-13) are each used to represent one value, where the first byte holds the least significant eight bits and the second byte holds the most significant eight bits.

2. Byte 18 was included in the Diskette Header for convenience and can be determined by adding 6 to the sum of the values in bytes 16 and 17.

3. Byte 19 was also included for convenience and can be determined by dividing 256 by the value in byte 18 and disregarding the remainder.

4. For compatibility reasons, User diskettes configured for variable length files contain the binary value of 9999 in bytes (12-13) because the maximum size of variable length files is the largest available size. Also, the bytes (20-21) on diskettes configured for fixed length files are not used because all files are of the same size.

The two hexadecimal dumps shown in Figures #F-2 and #F-3 are examples of a Diskette Header.

3. Directory (Starts at Track 1, Sector 3)

The length of the Directory is calculated by the Configurator program and is directly proportional to:

a. maximum number of files/diskette
b. maximum number of characters/File Name
c. maximum number of characters/Extension

Fixed Length Files

```
55 53 52 OF 05 00 08 02 01 40 00 1E 00 00 50 14
OC 26 06 OC DB 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

FIGURE #F-2

Variable Length Files

```
55 53 52 OF 05 00 88 02 01 40 00 OF 27 00 50 14
OC 26 06 1E 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

FIGURE #F-3

For each file that is stored on a User diskette, there exists one Directory entry in the Directory. An illustration of a Directory Entry is shown in Figure #F-4.

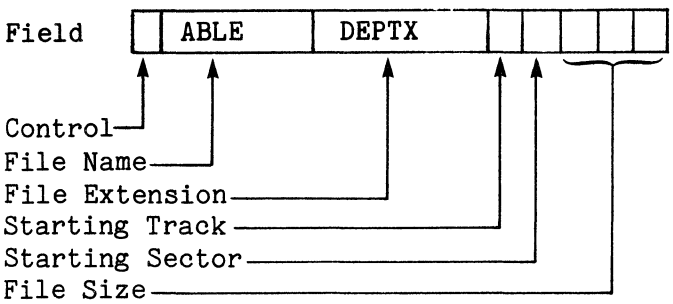


FIGURE #F-4

Bit	Value and Meaning
7(MSB)	0=Library slot is unoccupied 1=Library slot is occupied
6	1=Truncate pending (used by variable length files)
5	1=File is protected
4	Unused
2	Internal use
3,1, and 0	000=Standard Data File 001=BASIC Text File 010=Forms Data File 011=Binary Data File 100=RUN Packed File 101=BASIC Packed File 110=Reserved 111=BASIC Object File

The Control field consists of eight bits which are described as follows:

The other fields in each Directory Entry are explained below.

Field	Description
File Name	Contains the (ASCII) file name which is left-justified. The unused bytes are set to binary value 0.
File Extension	Contains the (ASCII) file extension which is left-justified. The unused bytes are set to binary value 0.
Starting Track	Contains the track (1-byte binary) number where the file begins.
Starting Sector	Contains the sector (1-byte binary) number where the file begins.
File Size	Contains the byte count (3-byte binary: LO,MO, HO) of the file.

The Directory is made up of one or more Directory Blocks; each Directory Block is two sectors long. Directory Blocks are generated by the Configurator program and contain Directory Entries.

Directory Print-out

```
UNUSED ENTRIES = 59

TEST                                  343
PROG 1                               1997  PAK
PROG 2                               1999  TXT
PROG 3                               1997  OBJ
BINTEST                                73  BIN
```

FIGURE #F-5

Fixed Length Files

```
80 54 45 53 54 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 02 01 57 01 00 89 50 52 4F 47 20 31 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 03 05 CD 07 00 81 50 52 4F
47 20 32 00 00 00 00 00 00 00 00 00 00 00 00 00
07 00 8B 50 52 4F 47 20 33 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 05 0D CD 07 00 83 42 49 4E 54 45 53 54
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 06 11 49 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 07
15 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

FIGURE #F-6

Variable Length Files

```
80 54 45 53 54 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 02 01 57 01 00 89 50 52 4F 20 31 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 02 04 CD 07 00 81 50 52 4F
47 20 32 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 02 14 CE
07 00 8B 50 52 4F 47 20 33 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 03 0A CD 07 00 83 42 49 4E 54 45 53 54
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 03 1A 49 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

FIGURE #F-7

(Directory Blocks provide Comm-Stor with an efficient means of searching the Directory.) The number of Directory Entries per Directory Block can be determined by dividing 256 by the number of characters per Directory Entry and disregarding the remainder. Figures #F-6 and #F-7 consist of a hexadecimal dump of a Directory Block and Figure #F-5 represents the corresponding Directory print-out.

F. Library

The starting location of the Library depends on the size of the Directory.

The Library contains the contents of each file on the diskette.

This page intentionally blank.

APPENDIX G

Comm-Stor IV ERROR MESSAGES

Number	Error Message	Description
0	SYSTEM	Indicates that the system has detected an equipment problem or a diskette with a bad Directory. If the error occurs again when another diskette is used, the operator should note the conditions which created the error and contact maintenance personnel.
-	?	An improper command has been entered. Example: .CM was entered instead of .CN
-	Bell	A character was entered at the terminal and sent to the modem when Data Set Ready was not present. This error is usually due to the operator forgetting to enter a period to symbolize the start of a command. Note: The bell signal is also used to indicate the completion of an Enter or Enter Automatic operation, or to alert the operator that a line is to be continued on the next display line.
1	NOT RDY	Indicates an attempt to access a drive when a diskette was either not inserted or improperly inserted, or an attempt to access Drive 2 in a single drive system.
2	DISKETTE	Indicates the system was unable to locate the proper location on a diskette where a file is stored or will be stored. The probable cause of this error is a bad diskette.
3	BAD READ	Indicates a file or part of a file could not be read without errors in eight attempts to read the file. The file should be rewritten on the diskette. If the error still occurs, that area of the diskette is probably bad.
4	PROTECT	Indicates an attempt to write on a protected diskette or file.
5	WRONG DK	Indicates the diskette is not a User diskette or it is a bad User diskette.
6	PREP SYS	Indicates the baud switch was not set to KYBD position when a Baud command was input.

Number	Error Message	Description
7	FULL DSK	Indicates the Directory is full. Either a file must be canceled from the diskette before entering a new file or a new diskette must be used.
8	NO FIND	Indicates a requested file does not exist in the Directory. Check to see that the file name and extension completely agree with the Directory entry.
9	ILLEGAL	An illegal operation has been attempted such as entering a file name longer than the configured maximum file name length.
10		(Reserved for future expansion.)
11	USR TABL	Indicates the system detected an improper command from the User Command Table. The User Command Table must be corrected using the Configuration diskette.
12	MODEM	Indicates an improper condition has been detected at the modem interface. One of the following conditions exist: Clear to Send was not asserted within 400 msec after Request to Send was asserted, or Data Set Ready was not asserted when attempting to perform a Send, Send Directory or Send Status Command.
13	NO ROOM	The program being resaved cannot fit into an available file slot. (The file is deleted from the diskette, but the program still remains in memory.)
14	OVERRUN	Input data in either the Enter or Receive Mode has exceeded the input rate of the system and data was lost. Example: A diskette search error has caused the data throughput of the system to be slowed down resulting in a data overrun condition. The probable cause is a bad diskette.
15		(Reserved for future expansion.)
16	AU	Indicates that the File Reference # has already been used. For example, the operator tried to open a file to a reference # already in use.
17	BB	Bad Buffer Size: The buffer size must be within the range 1-255.

Number	Error Message	Description
18	BD	Bad Drive #: Drive number must be 1 or 2.
19	BF	Bad File Operation: Example: trying to reposition in a mode other than random, or specifying an illegal file name in a CREATE or KILL command.
20	BN	Bad Reference #: The reference number must be within the range 0-24.
21	BR	Bad Record Size: Record sizes must be within the range of 1 to the total file size.
22	BS	Bad Subscript: Tried to reference a matrix element outside the dimensions of the array.
23	CN	Can't Continue: Tried to continue a program that does not exist, or tried to continue after a new line has been entered.
24	DD	Doubly Dimensioned Array: Tried to redimension a previously dimensioned array.
25	DE	Duplicate Entry: Attempt to create two files with the same name.
26	DZ	Divide by Zero: Tried to divide by zero, which is illegal.
27	EF	End of File: Reached End of File during a Print or Read operation.
28	FC	Illegal Function Call: Called a function with improper arguments.
29		(Reserved for Future Expansion)
30	FO	File Manager Buffer Overflow: Tried to assign memory space to a file in excess of available memory.
31	ID	Illegal Direct: This statement may not be used in the Direct Mode.
32	II	Improper Image: The image does not correspond to the data or the image itself contains a conflict.
33	IU	Illegal User Routine: Incorrect format of a USC command or USF function call.
34	LN	Line Number Out of Range: All line numbers must be between 0 and 64000.

Number	Error Message	Description
35	LS	String Too Large: Attempted to create a string longer than 255 characters.
36	NF	Next Without For: There is no FOR Statement corresponding to the next statement, or a branch was made in the middle of the FOR/NEXT loop.
37	NO	File Reference Number Not Opened: Did not open a file to this particular reference number.
38	OB	Not Open at Beginning: All files used in a program must be opened prior to a string operation.
39	OD	Out of Data: Ran out of data while trying to satisfy the Read Statement requirements.
40	OE	Option Error: An option (1 char letter) specified in the Create, Open, Assign, Merge or Save commands is an illegal character.
41	OM	Out of Memory: Ran out of available memory.
42	OV	Overflow: The result of a calculation was too large to be represented in the specified variable type. If the result of a calculation is too small to be represented, then the result will be zero.
43	PT	The file required for this operation is not in the correct format. Select another file or change the format of the required file.
44	RG	Return Without Prior GOSUB: A Return Statement was executed before a corresponding GOSUB, CALL, or RTCS INTERRUPTed.
45	SN	Illegal Syntax: The syntax in this line is incorrect.
46	ST	String Formula Too Complex: The string expression was too complex to be evaluated. This can be solved by breaking up the expression into two or more smaller ones.
47	TM	Type Mismatch: The variable type (i.e. string-integer-single-double precision) is not correct for this situation.
48	UF	Undefined Function: Tried to reference a user function that does not exist.
49	US	Undefined Statement: Tried to branch to a nonexistent line.

INDEX

Comm-Stor IV
PROGRAMMER MANUAL

READER'S COMMENTS

Your comments and suggestions will help us to maintain the quality and effectiveness of our publications. We will accept marked photocopies of those pages which you feel may require our further attention. Forward your comments to:

SYKES DATATRONICS, INC.
Dept. of Technical Publications
375 Orchard Street
Rochester, New York 14606

Does this manual satisfy the need you think it was intended to satisfy?

_____

Does this manual satisfy your needs? _____ Why? _____

_____
_____
_____

Would you please indicate any factual errors you have found.

_____
_____
_____

What faults do you find with this manual?

_____
_____
_____

General Comments:

_____
_____
_____
_____
_____

Please describe your position: _____

Name _____ Organization _____

Street _____ Department _____

City _____ State _____ Zip or Country _____

257A

FOLD and TEAR along perforation.

SYKES DATATRONICS INC. • 375 ORCHARD ST. • ROCHESTER, N.Y. 14606 • 716-458-8000 • TELEX: 97-8326 SYKES DATA ROC

Comm-Stor IV • Programmer Manual