Extended Disk
BASIC
User's Manual

Describes DBASIC, Release 1.1

Processor Technology
Corporation
7100 Johnson Industrial Drive
Pleasanton, CA 94566
Telephone (415) 829-2600

Copyright (C) 1978 by Processor Technology Corporation
Manual Part No 727037
All rights reserved.
IMPORTANT NOTICE

This manual, and the programs it describes, are copyrighted by Processor Technology Corporation. All rights are reserved. All Processor Technology software packages are distributed through authorized dealers solely for sale to individual retail customers. Wholesaling of these packages is not permitted under the agreement between Processor Technology and its dealers. No license to copy or duplicate is granted with distribution or subsequent sale.

TABLE OF CONTENTS

1. INTRODUCTION. ........................................ 1-1
  1.1 HOW TO USE THIS BOOK .............................. 1-2
  1.2 SYSTEM REQUIREMENTS .............................. 1-3

2. ELEMENTS OF BASIC USAGE ........................... 2-1
  2.1 HOW TO INITIALIZE AND LEAVE BASIC ............... 2-1
  2.2 DEMONSTRATION PROGRAMS ............................ 2-3
  2.3 DEFINITIONS OF COMMANDS AND STATEMENTS ........ 2-3
  2.4 DESCRIPTION OF BASIC STATEMENTS ................ 2-4
     2.4.1 Constants ................................... 2-5
     2.4.2 Variables ................................... 2-5
     2.4.3 Expressions ................................ 2-6

2.5 DEFINITION OF A PROGRAM .............................. 2-6
2.6 THE CALCULATOR MODE OF BASIC ..................... 2-7

3. HOW TO CREATE, EDIT, EXECUTE, AND SAVE A PROGRAM. .... 3-1
  3.1 CREATING A PROGRAM ................................ 3-1
  3.2 COMMANDS TO AID IN CREATING A PROGRAM .......... 3-2
     LIST, DEL, SCRATCH, REN, EDIT
     REN, CONT, CLEAR
  3.3 EXECUTING A PROGRAM ................................ 3-7
  3.4 HANDLING PROGRAM FILES ON DISKETTE ............... 3-9
     3.4.1 Text and Semi-Compiled Modes of Program Storage... 3-9
     3.4.2 Commands for Handling Diskette Program Files .... 3-10
        SAVE, GET, XEQ, APPEND, KILL, CRT

4. A BEGINNER'S SET OF BASIC STATEMENTS ............... 4-1
   REM, LET
   4.1 GETTING DATA INTO AND OUT OF THE PROGRAM ........ 4-2
       INPUT, PRINT
   4.2 RETRIEVING DATA FROM WITHIN A PROGRAM ............ 4-5
       READ, DATE, TYP(0), RESTORE, ON...RESTORE
       END, STOP, PAUSE
   4.3 STOPPING OR DELAYING EXECUTION .................... 4-9
   4.4 EXECUTION CONTROL ................................ 4-11
       GO TO, ON...GO TO, FOR and NEXT, EXIT,
       ON...EXIT
   4.5 EXPRESSION EVALUATION .............................. 4-16
       4.5.1 Arithmetic Operators ......................... 4-17
       4.5.2 String Operator .............................. 4-17
       4.5.3 Relational Operators ......................... 4-17
       4.5.4 Logical Operators ........................... 4-18
       4.5.5 Logical and Relational Operations in
       Algebraic Computations .......................... 4-18
       4.5.6 Evaluating Expressions in IF Statements .... 4-19

       IF
5. ADVANCED BASIC ....... 5
  5.1 SUBROUTINES ...... 5
      GOSUB, RETURN, ON...GOSUB 5
  5.2 FUNCTIONS ......... 4
      5.2.1 General Mathematical Functions.
      ABS, EXP, INT, LOG, LOGIC, RND, SQR, SGN
      5.2.2 Trigonometric Functions
      SIN, COS, TAN, AR
      DEG, RN
  5.3 CHARACTER STRINGS .... 9
      5.3.1 String Constants .... 9
      5.3.2 String Variables .... 10
      5.3.3 String Expressions.
      DIM, SEARCH
      5.3.4 String Functions .... 13
      Substring, LEN, ASC and CHR, VAL and STR, ERR(0)
  5.4 DIMENSIONED VARIABLES .... 13
      DIM, FR(0)
  5.5 USING DISKETTE FILES FOR DATA STORAGE.
      FILE, PRINT, READ, REWIND, CLOSE,
      PURGE, SET XI, SET PB, SET OF,
      EOF, TUOFF and TUON
  5.6 CONTROLLING THE FORMAT OF NUMERIC OUTPUT .... 36
      Formatted PRINT
  5.7 CONTROLLED INPUT .. 41
      Controlled INPUT
  5.8 ERROR CONTROL .... 42
      ERSET and ERCLR, ON...ERRSET, ERR(0), SYST(0)
  5.9 COMMANDS CAN BE STATEMENTS AND STATEMENT
      COMMANDS .... 44
      5.9.1 The SET Commands .... 44
      SET DS, SET LL, SET ML
      5.9.2 BYE and SCRATCH Commands .... 45
  5.10 CURSOR CONTROL .... 46
      CURSOR, POS(0)
  6. MACHINE LEVEL INTERFACE .... 6
  6.1 WRITING TO A PORT OR MEMORY LOCATION .... 6
      POKE and OUT, PEEK and INP, LOAD, CALL, WAIT

7. MATRIX OPERATIONS .... 7
    7.1 MATRIX INITIALIZATION .... 7
    7.2 MATRIX COPY .... 7
    7.3 SCALAR OPERATIONS .... 7
    7.4 MATRIX ARITHMETIC OPERATIONS .... 7
    7.5 MATRIX FUNCTIONS .... 7
    7.6 REDIMENSIONING MATRICES .... 7

APPENDICES

APPENDIX 1 EXTENDED BASIC COMMAND AND STATEMENT SUMMARY
      AND INDEX
APPENDIX 2 EXTENDED BASIC FUNCTION SUMMARY AND INDEX
APPENDIX 3 ERROR MESSAGES
APPENDIX 4 TABLE OF ASCII CODES (Zero Parity)
APPENDIX 5 HEXADECIMAL-DECIMAL INTEGER CONVERSION TABLE
APPENDIX 6 BIBLIOGRAPHY
1. INTRODUCTION

Extended BASIC is a special adaptation of BASIC (Beginner's all-purpose Symbolic Instruction Code) for use with the Processor Technology Disk Operating System (PTDOS) and Helios II Disk Memory System. The BASIC interpreter was selected for adaptation because it is simple and easy to learn while providing the powerful capabilities of a high-level language. Thus, it is ideal for the user who is a novice at using programming languages as well as for the advanced user who wants to work with subroutines, functions, strings, and machine-level interfaces.

Some of the outstanding features available in Extended BASIC are:

* Fully-formatted output to a variety of devices
* Many function subprograms, including mathematical, string, and video functions
* Program and data storage on floppy disk
* Full eight-digit precision
* User-defined functions on one or more lines
* Calculator mode for immediate answers
* Moving-cursor editing on video displays
* Complete capability for string handling
* Functions and statements for communicating with any number of input/output channels
* Ability to view memory locations, change values, and branch to absolute addresses
* DATA files
* Matrix functions including INVERT

BASIC is a conversational language, which means that you can engage in a dialog with BASIC by typing messages at a terminal and receiving messages from a display device. For example:

```
BASIC: READY

User: 10 PRINT "WHAT IS THE VALUE OF X" <CR> - The user enters
       20 INPUT X <CR> the lines of a program,
       30 LET Y = X^3 <CR> each followed by a
       40 PRINT "X CUBED IS ";X^3 <CR> carriage return.
       DEL 30 <CR> - The user deletes line 30.
       LIST <CR> - The user tells BASIC to
                      list what has been typed.
```
1.1. HOW TO USE THIS BOOK

This book is intended as a description of this particular version of BASIC, namely Extended Disk BASIC. Several useful beginning books are listed in Appendix 6 for those who need more background.

Read this book from cover to cover first, as a text. The material is presented in increasing difficulty from front to back. After you are familiar with Extended BASIC, you can use the book as a reference. In addition, statement and command summaries are given in Appendix 1. Appendix 2 is a function summary.

Section 2 gives background information needed for working with BASIC. It presents the fundamental definitions and modes of operation, and tells how to initialize and leave BASIC.

Section 3 describes the mechanics of writing BASIC programs, executing them, saving programs on diskette, and retrieving them at the appropriate time.

Section 4 describes an introductory set of statements, the instructions that make up a BASIC program. The statements described in section 4 are the simplest in the language, but they can be used to solve many math and business applications.

Section 5 is referred to as "Advanced BASIC," but do not be taken aback by the term "advanced." All of BASIC is, as the name implies, relatively simple to learn. Section 5 merely goes further into the language by teaching the use of subroutines and functions, how to work with strings of characters, saving data on diskette, and formatting output data.

Section 6 is for specialists. Those of you who have expanded your computer to send and receive data at a number of input/output ports will be interested in reading about the machine-level interfaces of BASIC.

Section 7 involves special statements, preceded by MAT, which involve the manipulation of matrices (two-dimensional arrays).

1-2 Disk BASIC

1.2. SYSTEM REQUIREMENTS

Extended BASIC must be used in conjunction with the Processor Technology Disk Operating System (PTDOS), software which is normally used on the Helios II Disk Memory System. Besides the hardware required to support PTDOS, BASIC requires additional memory from location 100 (Hex) to an address which depends on how BASIC is initialized. (See section 2.1.) During initialization, you will have the opportunity to delete portions of the BASIC interpreter that may not be of use to you, thereby saving the memory they would have occupied for program storage. The last address used by the BASIC interpreter is given during the initialization process, but an additional amount of memory is required for program and data storage. Overall, at least 24K is recommended.

The current BASIC program begins at the address immediately above BASIC itself, an address which varies with initialization as described above.
BASIC includes video display driver software. During initialization it is possible to select an option which sends all output from BASIC to another PTDOS file, which could be a device driver file, instead of to the video display driver. If the video display driver will be used, it is necessary to have compatible video display circuitry, normally the video display circuitry in the Sol Terminal Computer or a VDM-1 video display module. This circuitry is based around a 1K page of system memory addressed at CCB0, which holds the characters to be displayed.

SECTION 2

2. ELEMENTS OF BASIC USAGE

Before writing and working with BASIC programs, you have to know how to get into the BASIC environment and the rules for using BASIC. This section presents the fundamentals of BASIC usage.

2.1. HOW TO INITIALIZE AND LEAVE BASIC

Extended BASIC is provided on a floppy diskette in a version compatible with the Processor Technology Disk Operating System (PTDOS). It is assumed in the instructions below that you have a Helios II Disk Memory System or other computer which is using PTDOS. 24K of random access memory starting at address 0 is recommended.

Extended Disk BASIC is stored on diskette under the name DBASIC. Although BASIC can be used directly from this disk file, it is more convenient to load in DBASIC from diskette, perform some initialization procedures, and store the initialized version under a new file name for normal usage. A procedure file named MAKEBASIC is supplied on your BASIC disk to help accomplish this initialization. To use it, put a PTDOS system disk in unit 0 and the BASIC disk in unit 1, and follow these steps:

1) In PTDOS command mode, type:

   DO MAKEBASIC/1

   This will clear memory and load in the uninitialized DBASIC from diskette. BASIC will begin execution and a copyright notice will appear, and then the message: SIZING MEMORY. At this time BASIC scans the memory locations above BASIC to determine how much space is available for program and data storage.

2) After a brief delay, the message:

   LAST AVAILABLE MEMORY LOCATION (HEX) IS nnnn

   appears, where nnnn is a memory address in hexadecimal notation. If an address appears which is lower than expected, it may be due to a bad memory address or the existence of read only memory at location nnnn + 1.
3) Next, a question appears:

DELETE MATRIX OPERATIONS?

Now type Y for yes or N for no. If you type Y, the part of
BASIC which performs matrix operations will be removed from the
version you are initializing, making more memory available for
programs and data. If you type N, a new message will appear; go to step 5 below.

4. If you typed Y the following additional message will appear:

DELETE EXTENDED FUNCTIONS?

Again type Y or N to remove or not remove an additional part
of BASIC which performs trigonometric functions and certain
other extended functions. The following functions cannot be
used if Y is typed: SIN, COS, TAN, EXP, ATN, LOG, LOG10, and
the exponentiation operator **.

5. After Y or N is typed, this message appears:

PTDOS OUTPUT or INTERNAL VDM DRIVER? (I,P)

If you want all output generated by BASIC to go to BASIC's
video display driver software, type I. (See Section 1.2 for
the hardware requirements of this software.) If you want all
of BASIC's output to go to the PTDOS Command Interpreter console
output file, type P. (The file used for console output may be
changed with the PTDOS SETOUT command. See Section 1.8 of
the PTDOS Manual.) Once you are in BASIC, you can still change the
output file with the SET OF statement.

6. After I or P is typed, the following message will appear:

SAVE FROM yyyy TO xxxx

where yyyy and xxxx represent addresses in hexadecimal form.
Depending on how you answered the questions above, the newly
initialized form of BASIC will now occupy the addresses between
yyyy and xxxx. A working copy of BASIC (named BASIC) is now
IMAGED onto the disk in unit 0. Since BASIC is stored as an
image file, you can now enter BASIC by simply typing BASIC <CR>
in PTDOS command mode. Be sure to keep a safe copy of DBASIC

in case a version that is initialized differently is needed. If
want a different version, simply repeat the above procedure.
You can see the PTDOS commands which accomplished the initiali-
ization by typing the command PRINT VARBASIC/1 in PTDOS.

After you are done working in BASIC, you can exit to PTDOS
command mode by simply typing BYE <CR>. Any program that
was in BASIC, and options you may have selected, will remain
intact. Note that many PTDOS commands not marked [SAFE], and
other operations can write over BASIC in memory.

If you are in PTDOS command mode, you can return to BASIC,
provided it is still intact in memory, by typing:

EXEC 100 <CR>

using this mode of re-entry preserves any program that was in
BASIC, as well as various options you may have selected when
you were last in BASIC. If you re-enter BASIC by typing its
name as a command, you clear any program that was in BASIC, and
set various other options to their default values.

If you have a BASIC program stored as a PTDOS file, and you
want to load BASIC, and load and run the program at the same time,
use the one command:

BASIC file name

using the program's file name. The program will begin execution
automatically.

When system power is first applied, and you BOOTLOAD PTDOS,
a DD file called START.UP, containing a series of commands,
is automatically executed. If you want your system to "come up"
in BASIC, or you would like to run a BASIC program on bootload-
ing, you can enter an appropriate command line the START.UP
file.

2.2. DEMONSTRATION PROGRAMS

The BASIC disk contains demonstration programs which illustrate
the power of this version of BASIC and may be studied as
examples of advanced programming techniques, by LISTING them.
For more information type BASIC RUNME/L <CR>.

2.3. DEFINITIONS OF COMMANDS AND STATEMENTS

Whenever you type a line of text ending with a carriage return
in the BASIC environment, BASIC interprets it as a command or
as a statement. A command is an instruction that is to be exe-
cuted immediately, while a statement is an instruction that is to
be executed at a later time, probably in a sequence with
other statements.

BASIC differentiates between commands and statements by the
presence or absence of line numbers. A statement is preceded

2-2 Disk BASIC

2-3 Disk BASIC
by a line number. A command is not. Examples of command lines are:

LIST 10,90 <CR>
DEL 70 <CR>
BYE <CR>

Examples of statement lines are:

10 LET A = 100 <CR>
70 PRINT A, 27 <CR>
100 INPUT X,Y,C <CR>

You can enter more than one statement on a line by using the colon as a separator. For example:

10 LET X = 0 : GO TO 150

is the same as

10 LET X = 0
20 GO TO 150

When entering multiple statements on a line, precede only the first statement with a line number. For example:

100 INPUT A,B,C:LET X = A - B*C

A command or statement has a keyword that tells what is to be done with the rest of the line. In the examples above, the keywords are LIST, DEL, BYE, LET, PRINT, and INPUT. Keywords can be abbreviated by eliminating characters on the right and following the abbreviation with a period. For example, the following statements are equivalent:

10 PRINT X,Y
10 PRINT X.Y
10 PRINT X,Y
10 PRINT X,Y
10 PR. X,Y
10 PR. X,Y

The minimum number of characters allowed in the abbreviation is determined by the number of characters required to uniquely identify the keyword and by a hierarchy of keywords in statements or commands. Appendices 1 and 2 indicate the minimum abbreviations allowed for all command and statement keywords.

2.4. DESCRIPTION OF BASIC STATEMENTS

A statement is preceded by a line number which must be an integer between 1 and 65535. This line number determines the statement's place in a sequence of statements. The first word following the statement number tells BASIC what operation is to be performed and how to treat the rest of the statement. For example:

200 PRINT "THIS IS AN EXAMPLE"

Indicates what is to be printed.

Tells BASIC that a printing operation is to take place.

Indicates that this statement will be executed before statements with line numbers greater than 200 and after statements with line numbers less than 200.

Blanks do not affect the meaning of a statement in BASIC. That is, the following are equivalent statements:

20 GO TO 200
20 GOTO 200

BASIC automatically removes blanks from statements as you enter them. Blanks in strings (discussed later) are not altered.

BASIC statements specify operations on constants, variables, and expressions. These terms are discussed in the units below.

2.4.1. Constants

A constant is a quantity that has a fixed value. In Extended BASIC constants are either numerical or string. A numerical constant is a number, and a string constant is a sequence of characters.

A numerical constant can be expressed in any of the following forms:

Examples

Integer
1, 4000, 32543, -17

Floating point 1.73, -1123.01, 0.0004

Exponential
3.100E-5, 10E4, 230E-12

A string constant is indicated by enclosing a string of characters in quotation marks. For example:

"Illinois"
"The answer is"

Strings are discussed in more detail in section 5.

2.4.2. Variables

A variable is an entity that can be assigned a value. In Extended BASIC a variable that can be assigned a numerical value has a name consisting of a single letter or a single letter followed by a digit. The following are examples of numerical values being assigned to numerical variables:

\[ A = 17 \]
\[ B9 = 147.2 \]

A variable that can be assigned a string value has a name consisting of a single letter followed by a dollar sign or a single letter followed by a digit followed by a dollar sign $.
Examples of stringy values being assigned to string variables are:

A$ = "J. PAUL JONES"
X$ = "711 N. Murry"
R98 = "Payables, Dec. 9"

2.4.3. Expressions

An expression is any combination of constants, variables, functions, and operators that has a numerical or string value. Examples are:

X^2 + Y = A+B
22 + A
"KON" + A$
NOT N

A numerical expression is an expression with a numerical value. It may include any of the following arithmetic operators:

\* exponentiate
\* multiply
/ divide
+ add
- subtract

However, a negative number cannot be raised to a power ((-x)^n) since the result could be a complex number. In an expression arithmetic operators are evaluated in the order shown below:

highest - (unary negate)
next highest ~
next highest * and /
lowest + and -

Expressions in parentheses are evaluated before any other part of an expression. For example:

A / 2 * B = (4 / C)^2
(third) [first]
[fourth] [second]
[fifth]

Numerical expressions can also include logical and relational operators. These are introduced in section 4.

Operations in string expressions are described in section 5.

2.5. DEFINITION OF A PROGRAM

A program is a stored sequence of instructions to the computer. The instructions are specified in statements arranged to solve a particular problem or perform a task. The statement numbers determine the sequence in which the instructions are carried out. For example, the following program averages numbers:

10 PRINT "HOW MANY NUMBERS DO YOU WANT TO AVERAGE?"
20 INPUT N
30 PRINT "TYPE ",N;"NUMBERS"
40 FOR I = 1 TO N
50 INPUT X
60 S = S + X
70 NEXT I
80 PRINT "THE AVERAGE IS ", S/N

2.6. THE CALCULATOR MODE OF BASIC

In unit 2.2, a statement was described as a user-typed line preceded by a statement number and a command was described as a user-typed line without a statement number. In Extended BASIC you can also type a statement without a statement number and it will be treated as a command. That is, BASIC executes the statement as soon as you type the carriage return at the end of the line. For example:

User: PRINT "5.78 SQUARED IS ",5.78^2 <CR>
BASIC: 5.78 SQUARED IS 33.3084

Thus, you can use BASIC as a calculator to perform immediate computations.

If you perform a sequence of operations in calculator mode, BASIC will remember the results of each statement just as it does in a program. For example:

User: LET A = 28.78 <CR>
INPUT X
BASIC: ? 2 <CR> The user types 2 in response to the ?.
User: LET B = A*X <CR>
IF B > X THEN PRINT B
BASIC: 41.56

In the documentation of individual statements in sections 4 and 5, statements that can be used in calculator mode are marked CALCULATOR in the box containing the statement form.

Without exception, all number in BASIC are decimal. This includes not only data values in constants, variables, and expressions, but the operands of BASIC statements and commands when they call for numeric values.
SECTION 3

3. HOW TO CREATE, EDIT, EXECUTE, AND SAVE A PROGRAM

A BASIC program is a stored sequence of instructions to the computer. This section tells how to enter a program into the computer, view the text of the program and alter it, execute the program, save it for future use, and retrieve it from storage.

3.1. CREATING A PROGRAM

To create a program, simply type the statements of the program in BASIC. Precede each statement with a statement number and follow it with a carriage return. For example:

User: 10 INPUT X,Y,Z <CR>
      20 PRINT X+Y+Z <CR>

A program now exists in BASIC. When executed the program will accept three numbers from the terminal and then print their sum.

When entering statements be careful not to create lines that will be too long when formatted by BASIC. BASIC will expand abbreviated statements; for example P. will become PRINT in a listing or edit. BASIC will insert blanks to improve readability, if the program was typed without them. These two factors can expand a line beyond the limit set by the SET LL= length command or statement. For more information about line length errors, see "LL" in Appendix 3.

It is not necessary to enter the statements in numerical order. BASIC will automatically arrange them in ascending order. To replace a statement, precede the new statement with the statement number of the line to be replaced. For example:

User: 20 INPUT X,Y <CR>
      10 PRINT "TYPE X AND Y" <CR>
      30 PRINT X*Y <CR>

      10 PRINT "TYPE X AND Y"  
      20 INPUT X,Y 
      30 PRINT "THE PRODUCT IS ",X*Y <CR>

      LIST <CR>

      PRINT "THE PRODUCT IS ",X*Y <CR>

The user enters the statements out of sequence. BASIC orders the statements and keeps only the last line entered for a given statement number.
While entering statements or commands in BASIC, you can use any of the following keys on the terminal to correct the line being typed:

- **DEL**: Deletes the current character and shifts the remainder of the line to the left.
- **← (Left Arrow)**: Moves the cursor one position to the left.
- **→ (Right Arrow)**: Moves the cursor one position to the right.
- **REPEAT**: Moves the cursor rapidly through the line when used with the left or right arrows. Also causes repetition of any key held down at the same time.
- **MODE**: Aborts a running program, infinite loop, listing, and getting or saving operations.
- **SELECT**: Deletes a line being typed. If used to stop a running program, all open files will be closed.
- **RETURN**: Terminates the line. The line remains as it appeared when the RETURN key was typed.
- **LINE FEED**: Terminates the line. All characters to the right of the cursor are erased.
- **↑ (Up Arrow)**: Initiates the insert mode. When you type characters in the insert mode, they are inserted at current cursor position, and the rest of the line is moved to the right.
- **↓ (Down Arrow)**: Terminates the insert mode.
- **CONTROL-X**: Cancels the line being typed, and positions the cursor on a new line. The cancelled line remains on the screen. May also be used while the user is typing a response to an INPUT statement in a running program.

If a control character (like CONTROL-X above) is typed at the beginning of a line on the video display or terminal, it will be displayed on the screen or terminal, and will be ignored by BASIC.

3.2. COMMANDS TO AID IN CREATING A PROGRAM

The commands described in this section are likely to be used while creating a program. The LIST command displays the program. DELETE and SCRATCH are used to erase statements. REN lets you automatically renumber statements. The EDIT command makes the line editor available.

### LIST Command

**General forms:**

- **LIST**: List the entire program.
- **LIST n**: List statement number n.
- **LIST n1,n**: List statement number n1 through the end of the program.
- **LIST n1,n2**: List all statements from the first through statement number n2.
- **Last in a series of statement numbers**
- **First in a series of statement numbers**

**Examples:**

- LIST 100,150 <CR>
- LIST 50, <CR>

The LIST command displays the indicated statements in increasing numerical order. It automatically formats the text of the statements, indenting and adding spaces where appropriate.

For example:

User:  10 FOR I = 1 TO 100 <CR>
       30 NEXT I <CR>
       20 PRINT I^2 <CR>
       LIST <CR>
       10 FOR I=1 TO 100
       20 PRINT I^2
       30 NEXT I

You can control the display of material using the following keys:

- **MODE key**: Aborts listing
- **Space bar**: Causes a pause in the listing. Striking it again causes the listing to resume.
- **1 through 9**: Changes the speed at which material is displayed.
DEL Command

General forms:

DEL all statements.
DEL n Delete statement number n.
DEL nl, Delete all statements from nl through the end of the program.
DEL ,n2 Delete all statements from the first through statement n2.
DEL nl,n2 Delete statement numbers nl through n2.

Examples:

DEL 150 <CR>
DEL 75,90 <CR>

The DEL command deletes the indicated statements. For example:

User: 100 LET A = 100 <CR>
110 INPUT X,Y,Z <CR>
120 PRINT (X+Y+Z)/A <CR>
DEL 110, <CR>
LIST <CR>
BASIC: 100 LET A = 100

Also, entering a line number that is not followed by a statement deletes a line. For example:

USER: 100 <CR>
LIST 100 <CR>
BASIC: Line 100 has been deleted.

SCRATCH Command

General form:

SCRATCH Delete the entire program and clear all variable definitions.

Examples:

SCR <CR>
SCRATCH <CR>

The SCRATCH command deletes the entire program and clears all variable definitions established during previous program runs or by statements executed in the calculator mode. For example:

User: A = 100 <CR>
PRINT A <CR> 100
SCR <CR>
PRINT A <CR> A
LIST <CR>

A receives a value of 100.
BASIC prints the assigned value for A. The SCR command clears variables.
A's value is now 0.
The SCR command has deleted all statements previously existing in the BASIC environment.

REN Command

General forms:

REn Renumber all statements. The first statement will be numbered 1 and subsequent statement numbers will be increments of 1.
REn n Renumber all statements. The first statement will be numbered n and subsequent statement numbers will be increments of n.
REn n,i Renumber all statements. The first statement will be numbered n and subsequent statement numbers will be increments of i.

Statement integer increment

Examples:

REN <CR>
REN 100,5 <CR>

The REN command renumbers all statements of the program as indicated, maintaining the correct order and branches in the program. For example:

User: 10 INPUT A,B <CR>
20 PRINT "A*B IS ",A*B <CR>
30 GO TO 10 <CR>
REN 100 <CR>
LIST <CR>
100 INPUT A,B
110 PRINT "A*B IS ",A*B
120 GO TO 100

Notice in line 120 that GO TO 10 has been changed to GO TO 100. If line 30 had been GO TO 58, thus referring to a line number which does not exist in the program to be renumbered, GO TO 58 would have been changed to GO TO 0, and an error message would have been printed. All references to non-existent line numbers will be changed to 0 before an error message is given.
EDIT Command

General form:

EDIT n  Edit statement number n.

Example:

EDIT 150 <CR>

The EDIT command displays the line to be edited on the line in which the EDIT command was typed, and the following line. EDIT then enters a mode that allows changes to the line using any of the following special keys:

<table>
<thead>
<tr>
<th>Key</th>
<th>Effect in EDIT Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEL</td>
<td>Deletes the current character and shifts the remainder of the line to the left.</td>
</tr>
<tr>
<td>← (Left Arrow)</td>
<td>Moves the cursor one position to the left.</td>
</tr>
<tr>
<td>→ (Right Arrow)</td>
<td>Moves the cursor one position to the right.</td>
</tr>
<tr>
<td>XREPEAT</td>
<td>Moves the cursor rapidly through the line when used with a ← or →.</td>
</tr>
<tr>
<td>CONTROL-X</td>
<td>Cancels the line being typed, and positions the cursor on a new line. The cancelled line remains on the screen.</td>
</tr>
<tr>
<td>MODE SELECT</td>
<td>Terminates the edit leaving the line as it was.</td>
</tr>
<tr>
<td>RETURN</td>
<td>Terminates the edit leaving the line as it appears on the screen.</td>
</tr>
<tr>
<td>LINE FEED</td>
<td>Terminates the edit deleting all characters to the right of the cursor.</td>
</tr>
<tr>
<td>↑ (Up Arrow)</td>
<td>Initiates the insert mode. When you type characters in the insert mode, they are inserted at the current cursor position and the rest of the line is moved to the right.</td>
</tr>
<tr>
<td>↓ (Down Arrow)</td>
<td>Terminates the insert mode.</td>
</tr>
</tbody>
</table>

3.3. EXECUTING A PROGRAM

When a program is executed with the RUN command, BASIC interprets each of the statements sequentially, then it carries out the instructions.

If BASIC encounters a problem during any of these steps, it prints a message describing the error. The meanings of BASIC error messages are given in Appendix I.

During execution a program can be interrupted by pressing the MOE key. This is true whether the program is running correctly, is in a loop, or is waiting for input. No information is lost and you can continue execution by giving the CONT command.

RUN Command

General forms:

RUN n  Execute the current program.

Example:

RUN <CR>

The RUN command executes all or part of the current program. If no statement number is specified, the command clears all variables and then executes the program. If a statement number is indicated, the command executes the program beginning with that statement number, but does not clear the variable definitions first. For example:

RUN 100 <CR>
User: 10 LET A = 10, B = 20, C = 30 <CR>
20 PRINT A*2*B-C <CR>
30 STOP <CR>
40 PRINT A/2*(B-C) <CR>
RUN <CR>

BASIC: 1970
STOP IN LINE 30
The STOP statement interrupts the program.

User: RUN 46 <CR>
BASIC: -1000
READY
Notice that A, B, and C still have the values assigned in statement 10.

The Mode Select key may be used to abort a running program.
When a program run terminates for this or any reason, all open files are closed.

**CONT Command**

- **General form:**
  
  CONT

  Continue execution.

- **Example:**

  CONT <CR>

The CONT command continues the execution of a program that was interrupted by the MODE key or stopped by the execution of a STOP statement (STOP is documented on page 4-9.) For example:

User: RUN <CR>
The user executes a program that computes and prints the squares of numbers 1 through 100.

BASIC: 10
4
9
16

User: MODE
BASIC: STOP IN LINE 70
The user presses the MODE key to interrupt execution.

User: CONT <CR>
The CONT command continues execution of the program.

BASIC: 75
36
49

If you edit any part of a program after interrupting execution, all variable definitions are lost. Thus you cannot stop a program's execution, change a statement in that program, and then continue execution or print variable names. When a program run is terminated for any reason, all open files are closed, which also could interfere with subsequent CONTinuation.

**CLEAR Command**

- **General form:**
  
  CLEAR

  Erases the definitions of all variables and leaves the program intact.

- **Example:**

  CLEAR <CR>

The CLEAR command clears all variable definitions but does not erase the statements of the current program. If CLEAR is used as a statement, all open files will be closed.

For example:

User: 10 A=10,B=20,C=30 <CR>
20 STOP <CR>
30 PRINT A,B,C <CR>
RUN <CR>

BASIC: STOP IN LINE 20
User: RUN 30 <CR>
BASIC: 10
20
30
READY
User: CLEAR <CR>
BASIC: 0
0
0
READY
User: LIST <CR>
BASIC: 10 A=10,B=20,C=30
20 STOP
30 PRINT A,B,C

The variables have the values assigned in line 10.

Variable definitions have been cleared.

The program remains intact.

### 3.4. Handling Program Files on Diskette

Once you have created and tested a program you can save it on diskette for future use. The commands described in this section can be used to save the program on diskette, read it as a file, read and automatically execute it, or read the program and append it to the statements currently in BASIC. Additional commands allow you to kill files or make a listing of all files of a specified type.

#### 3.4.1 Text and Semi-Compiled Modes of Program Storage

Four commands involved in storing and retrieving programs from diskette: SAVE, GET, APPEND, and XEQ, all have optional parameters T, for text mode of storage, or C, for semi-compiled mode of storage. (APPEND does not offer the semi-compiled option.) In text mode, the current program is saved literally, as the program would appear when listed. If a program may be used with other versions of BASIC, or other editors, it should
be saved in this form. In semi-compiled mode, the program is partially compiled, and is stored on diskette in a condensed form which saves space, and allows programs to be recorded and accessed faster. The semi-compiled program may not be intelligible to other versions of BASIC, however, and cannot be manipulated in a meaningful way by other editors.

3.4.2 Commands for Handling Diskette Program Files

Most of the commands for manipulating diskette program files, which are described below, use the following general form:

**COMMAND file name**

The file name is the name of a PTDOS file, and subject to the same conditions as any other PTDOS file. The file name can be from one to eight alphanumeric characters. Two extra characters may be added to the end of the file name (making the name up to 18 characters long) which specify the diskette drive unit to be used in the command. Hello II Disk Memory systems can be configured with up to eight units, each of which can contain a diskette. The minimum, and most popular system, has two units called 0 and 1. The characters to add are the slash (/) and a digit from 0 to 1 or 0 to 7, depending on how many units your system has. Note that blanks are not allowed: MYFILE/1 is an acceptable filename. Note that the unit specified does not refer to a particular diskette, but to any diskette which is inserted in the specified unit. If the diskette containing MYFILE were in unit 0, it would be called MYFILE/0, or just MYFILE (if 0 is the default unit). If the same diskette were inserted in unit 1, the same file would be called MYFILE/1.

If the file name is given alone in the command, without a unit specification, the default unit (usually 0) is used. PTDOS allows the user to change which unit will be the default unit.

In all, BASIC can create and manipulate files of four types. Besides types C and T (see above), data files use types R and S. The commands in BASIC for handling these four types, use the first letters T, C, R, and S to specify the types. BASIC uses PTDOS system calls in working with these files, and all four types will appear in the PTDOS directory of the diskette on which they are written, and will be listed by the PTDOS FILES command. However, the file type designation in the PTDOS directory is different from the type letters used by BASIC. Below is a table which relates the two type designations:

<table>
<thead>
<tr>
<th>BASIC Type</th>
<th>PTDOS Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>06</td>
<td>Text-type program files</td>
</tr>
<tr>
<td>C</td>
<td>05</td>
<td>Semi-compiled program files</td>
</tr>
<tr>
<td>R</td>
<td>08</td>
<td>Random access data files</td>
</tr>
<tr>
<td>S</td>
<td>07</td>
<td>Serial access data files</td>
</tr>
</tbody>
</table>

With certain limitations, a file of any type created in BASIC may be manipulated from within PTDOS, and vice versa. In particular, it is convenient to create or edit BASIC programs in text form with the PTDOS editor. This editor has features which are not available in the BASIC editor. The line length of such programs should be limited to 64 characters, however, to avoid truncation.

If a PTDOS file contains the appropriate contents, BASIC can be used to manipulate the file, no matter what type is assigned.

**SAVE Command**

The SAVE command writes the current program on a disk file and labels the file with the specified name. If the diskette already contains a file of the specified name, that file will be overwritten. SAVED files are "information-protected". They will not by shown by the simple PTDOS FILES command.

The T and C options let you specify that the text of your program is to be saved or that a semi-compiled version of the program is to be saved. C (semi-compiled) is the default option and need not be specified. If a file of the given name already exists on diskette, but is of a different type (T or C) than given, an error message will be displayed. Semi-compiled program files are name- and attribute-protected as well as information-protected.

In deciding whether to save your program in text or in semi-compiled form, keep the following factors in mind:

<table>
<thead>
<tr>
<th>Semi-compiled</th>
<th>versus</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Is more efficient</td>
<td>-Is recognizable as a sequence of BASIC statements</td>
<td></td>
</tr>
<tr>
<td>-Loads more quickly</td>
<td>-Can be edited by editors outside BASIC</td>
<td></td>
</tr>
<tr>
<td>-Can be saved more quickly</td>
<td>-Might be dependent on the version of BASIC in use</td>
<td></td>
</tr>
<tr>
<td>-Cannot be edited by external editors</td>
<td>-Is independent of the version of BASIC in use</td>
<td></td>
</tr>
</tbody>
</table>

For programs you intend to preserve and use frequently, it is best to save them in both modes: in text mode to preserve complete documentation and enable compatibility with other editors, and in semi-compiled form for rapid loading.
For example:

User:  10 PRINT "ENTER INTEREST RATE" <CR>
20 INPUT R <CR>
25 S = 1 <CR>
30 FOR I = 1 TO 100 <CR>
40 S = S + S*R <CR>
50 IF S >= 2 THEN 70 <CR>
60 NEXT I <CR>
70 PRINT "INVESTMENT DOUBLES IN ",I," YEARS" <CR>
SAVE INV <CR>

BASIC: (Records the program on diskette)
READY

GET Command

<table>
<thead>
<tr>
<th>General form:</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET file name</td>
</tr>
<tr>
<td>1 to 8 -------</td>
</tr>
</tbody>
</table>

Examples:

GET SUMS <CR>
GET AB3J/2 <CR>

The GET command searches the directory for the specified file, then reads the file making the program contained on it available in BASIC. Any statements residing in BASIC before the file was read are lost. The GET command determines whether the file was SAVED in text or semi-compiled form and acts accordingly.

The command below retrieves a program file named FAC from unit 1

GET FAC/1

An example of an interchange using the GET command follows:

User: LIST <CR>  BASIC generates no listing--there are no statements residing in BASIC.

GET INV <CR>  BASIC: (Reads the file from diskette)
READY

User: LIST <CR>

BASIC: 28 PRINT "ENTER INTEREST RATE"
28 INPUT R <CR>
25 S=1 <CR>
30 FOR I=1 TO 100 <CR>
40 S=S+S*R <CR>
50 IF S >= 2 THEN 70 <CR>
60 NEXT I <CR>
70 PRINT "INVESTMENT DOUBLES IN ",I," YEARS"

3-12  Disk BASIC

XEQ Command

General form:

<table>
<thead>
<tr>
<th>XEQ file name</th>
<th>Read the specified file from diskette and execute the program contained in it.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 8 -------</td>
<td>characters/unit</td>
</tr>
</tbody>
</table>

Examples:

XEQ SCR <CR>
XEQ STRAIGHT/1 <CR>

The XEQ command reads the specified file, making the program contained on it available in BASIC, and begins execution. Any statements residing in BASIC before the file was read are lost.

For example:

User: XEQ INV <CR>

BASIC: ENTER INTEREST RATE

BASIC begins execution of the program contained on file INV.

APPEND Command

General form:

<table>
<thead>
<tr>
<th>APPEND file name</th>
<th>Read the specified file from disk and merge the program contained on it with the statements already residing in BASIC.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 8</td>
<td>characters/unit</td>
</tr>
</tbody>
</table>

Example:

APPEND PROG2/1

The APPEND command searches the directory for the named file. Without erasing the statements currently in BASIC, it reads the file and merges the statements found there with the existing statements. The line numbers of statements from the appended file determine their positions with respect to the statements already in BASIC. If a line number from the file is the same as that of a statement residing in BASIC, the statement from the file replaces the previous statement. If no unit is specified in the command, the default unit is used.

Note: Only text files can be appended.

For example:
User: LIST <CR>
BASIC: 10 LET X=0
20 PRINT "ENTER Y AND Z"
30 INPUT Y,Z
User: APPEND PART2 <CR>
BASIC: (Reads the file from diskette)
READY
User: LIST <CR>
BASIC: 10 LET X=0
20 PRINT "ENTER Y AND Z"
30 INPUT Y,Z
100 A1=X+Y+Z
110 A2=X-Y-Z
120 A3=X-Y+Z
130 PRINT A1,A2,A3

KILL Command

General Form:
KILL file name
    Kill the named file, as if it were
    erased.

1 to 8 characters[/unit]

Examples:
KILL JEC
KILL USELESS/1

KILL performs operations that may be thought of as "erasing" the
named file: the file name is removed from the directory, and the
space used by the file is returned to the Free Space Map for use
by other files. (Actually the file is not literally erased, but
the space used is made available for re-use.)

CAT Command

General Form:
CAT [/unit] {type} Displays a catalog of files of the
the specified type on the specified
diskette drive unit.

Examples:
CAT
CAT /1
CAT /4T

CAT reads the directory of the specified unit, or the default
unit if none is specified, and prints a listing of files of the
specified type. There are four possible types:

T  Text type program files
C  Semi-compiled program files (default)
S  Serial data files
R  Random data files

If no type is specified, files of type C only will be listed.
The PFSOS FILES command performs a function similar to the
BASIC CAT command, but it also allows all files on a given disk-
ette to be listed in one command.

Text-type program files and serial access data files can be
created outside BASIC and used within BASIC provided that
program lines are not longer than 132 characters. However, to
be CATalogued, a program file must be given the type 86 and a
serial access data file must be given the type 07. Semi-
compiled program files and random access data files may be
created only within BASIC.
SECTION 4

4. A BEGINNER'S SET OF BASIC STATEMENTS

You can write BASIC programs for a multitude of mathematical
and business applications using just the statements described
in this section. This section tells how to assign values to
variables, perform data input and output, stop a program,
control the sequence in which statements are executed, and make
logical decisions. These include the simpler BASIC concepts.
After you have become familiar with the statements presented in
this section, read Section 5 to learn about the more extended
BASIC concepts.

REM Statement

General Form:

REM {any series of characters} Has no effect on program
execution.

Examples:

10 REM
100 REM: THIS PROGRAM COMPUTES INCOME TAX

The REM statement allows you to insert comments and messages
within a program. It is a good practice to include remarks
about the purpose of a program and how to use it. For example:

10 REM - THIS PROGRAM COMPUTES THE TOTAL INTEREST
20 REM - ON A TEN-YEAR LOAN
30 REM
40 REM - TO USE IT YOU MUST Supply THE PRINCIPAL
50 REM - AND THE INTEREST RATE
60 REM
70 PRINT "ENTER THE PRINCIPAL"
80 INPUT P


200 PRINT "THE INTEREST IS "; I
LET Statement

General forms:

\( \text{LET} \) \ variable \ = \ expression \ Assigns \ the \ value \ of \ the
\( \text{LET} \) \ variable1 \ = \ expression1, \ variable2 \ = \ expression2, \ ...

expression \ to \ the \ variable.

Examples:

10 LET X = 100.50
100 A1=12.7, A2=5.4, A3=50
200 LET X$ = "SHARKEYPORT"

The LET statement evaluates an expression and assigns its value to a variable. The variable may be a numeric or string variable and the value of the expression can be a number or a character string. The value of the expression and the variable must be the same type. For example:

10 LET A=0, B=100, C$="FIRST"
20 PRINT A, C$ 
30 A = A + B, C$ = 'SECOND'
40 PRINT A, C$

The equal sign is not a mathematic operator. It is an assignment operator. Thus \( A = A + B \) assigns to \( A \) the previous value of \( A \) plus the value of \( B \). The word \"LET\" is optional; \( \text{LET} \ X=1 \) is equivalent to \( X=1 \).

4.1. GETTING DATA INTO AND OUT OF THE PROGRAM

A program must read and write information to communicate with a user. Using the INPUT and PRINT statements is the simplest way to have your program perform input and output.

The INPUT statement reads data typed at the terminal. The form of the PRINT statement described below displays information at the terminal's display device. Using these two statements, you can make your program converse with a user at the terminal.

INPUT Statement

General forms:

\( \text{INPUT} \) \ variable, \ var2, \ ... \ Reads \ one \ or \ more \ values \ from \ the
\( \text{INPUT} \) \ \text{variable} \ \mid \ \text{message}, \ var1, \ var2, \ ... \ Prints \ the \ message, \ then
\( \text{message} \) \ \text{any characters} \ \mid \ \text{variable} \ \text{and assigns them to \( \text{var1}, \ \text{var2}, \ etc.} \)

Examples:

10 INPUT X
100 INPUT "WHAT IS THE VALUE OF S",S
200 INPUT, A1, A2, A3, N, T$, Y

The INPUT statement accepts one or more values entered at the terminal and assigns them in order to the specified variables. The values entered must agree with the type of variable receiving the value.

When an INPUT statement is executed, BASIC requests values from the terminal by printing a question mark or the message, if you have specified one. You may enter one or more values after the question mark, but not more than are required by the INPUT statement. If you enter several values on one line, they must be separated by commas. BASIC prompts for additional values with two question marks until all values required by the INPUT statement have been entered. For example:

10 PRINT "ENTER VALUES FOR A, B, C, & D "
20 INPUT A,B,C,D
30 PRINT "A*B/C*D IS ",A*B*C*D

When executed, this program accepts data from the terminal as follows:

User: \text{RUN} <CR>
BASIC: \text{ENTER VALUES FOR A, B, C, & D} \text{75.7, 7.4 <CR>}
User: \text{75.7, 7.4 <CR>}
BASIC: \text{75.7, 7.4 <CR>}

When a message is included in the INPUT statement, that message is displayed as a prompt before data is accepted from the terminal. For example:

User: \text{10 INPUT "WHAT IS YOUR NAME? ",NS <CR>}
20 PRINT "HI ",NS <CR>
RUN <CR>
BASIC: \text{WHAT IS YOUR NAME?} \text{SUE <CR>}

The user types "SUE in response to the prompt.
If the message used is "" (no message) then the normal ? prompt is surprised.

If a comma is placed in the statement after the word INPUT, then the carriage return and line feed will be surprised when the user depresses the carriage return key. In this way the next message printed by BASIC may appear on the same line. The program below illustrates this feature:

User: 10 INPUT "GIVE A VALUE TO BE CALLED: ",A
20 PRINT "A": A =; A*A
RUN <CR>

BASIC: GIVE A VALUE TO BE CALLED: 3 * 3 = 9

The user typed only 3 <CR> as input; BASIC completed the line.

If an INPUT statement requests input for a numeric variable, and the user's response contains an inappropriate character, the message INPUT ERROR, RETYPE appears, and the user is given another chance to type appropriate values. If the ERRSET statement is in effect, no message is given, but an IN error message is made available through the ERR(0) function.

**PRINT Statement**

**General forms:**

<table>
<thead>
<tr>
<th>PRINT</th>
<th>Skips one line.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINT exp</td>
<td>Displays the value of exp.</td>
</tr>
<tr>
<td>PRINT expl, exp2, ...</td>
<td>Displays the values of expl, exp2, etc., each filling 14 columns.</td>
</tr>
<tr>
<td>PRINT expl; exp2; ...</td>
<td>Displays the values of expl, exp2, etc.</td>
</tr>
</tbody>
</table>

**Examples:**

10 PRINT X
100 PRINT "THE SUM IS ";A+B+C+D
200 PRINT X,Y,Z;A,B/X;LS

The PRINT statement displays information at the terminal. The information displayed is the value of each expression. It is displayed in order and the separation between one value and the next is determined by the separator used. If a comma is used as a separator, each value is printed at the left of a field of 14 character positions. If a semicolon is used between two expressions, the second is printed one space after the first.

For example:

User: 10 PRINT 5, 10, 15; 20 <CR>
RUN <CR>

BASIC: 5 10 15 20

The output from each PRINT statement begins on a new line unless the statement ends with a separator. In this case, the next PRINT statement will cause values to be displayed on the same line and the separator will determine the position at which the cursor (or print head) will remain. For example:

User: 5 LET A1 = 1, A2 = 2, A3 = 3, A4 = 4 <CR>
10 PRINT A1,A2; <CR>
20 PRINT A3,A4 <CR>
30 PRINT "NEXT LINE" <CR>
RUN <CR>

BASIC: 1 2 3 4

NEXT LINE

READY

The following expressions can be used in a PRINT statement for further control over the position of output:

**TAB(exp)** Causes the cursor to move horizontally to the character position given by the value of exp (any numerical expression.) This function may only be used in a PRINT statement.

"\c" Prints the control character c. Printing certain control characters performs a function on the video display. Note that the character is preceded by a back slash (\). A few of the special control characters and their functions are:

- Control M = Carriage return
- Control J = Line feed
- Control K = Home cursor and clear screen
- Control W = Home cursor

Printing other control characters displays special symbols on the screen, as shown in the Sol manual.

"\n" Print a single backslash character (\).

For example:

10 PRINT TAB(I);"DECIMAL";TAB(I+30);"ENGLISH"
100 PRINT X, "\{\}Y, \{\}Z

Statement 10 prints ENGLISH 30 columns beyond DECIMAL; 100 prints the values of X, Y, and Z, each on a new line.

While the PRINT statement is executing and values are being output, it is possible to interrupt the printing by depressing the space bar on the keyboard. Depressing the space bar a second time will cause printing to resume. The speed of printing may be controlled with the number keys 1 through 9, key 1 giving the fastest speed. The SET DS = nexpr command also controls speed of printing to the video display, but has the additional effect of controlling all output to the screen, whether or not it was generated by a PRINT statement. More complex forms of the PRINT statement are covered in Section 5.5 and 5.6.
4.2. RETRIEVING DATA FROM WITHIN A PROGRAM

You can place data in a BASIC program using the DATA statement and access it as needed using the READ statement. The RESTORE statement allows you to start reading data again from the first DATA statement or from a specified DATA statement. The TYP() function allows you to determine the type of data to be read from the DATA statement corresponding to the next READ statement.

Data may also be stored as diskette data files. This subject is covered in Section 5.5.

READ Statement

General form:

```
READ var1, var2, ...    Reads one or more values from DATA
variable               statements and stores them in var1,
                       var2, etc.
```

Examples:

```
10 READ X2
100 READ A1, A2, A3, M$```

The READ statement reads one or more values from one or more DATA statements and assigns the values to specified variables. The value read must be the same type as the variable it is assigned to.

An example of a program using the READ statement is shown in the explanation of the DATA statement.

DATA Statement

General form:

```
DATA constant1, constant2, ... Specifies one or more
number     values that can be read by an READ statement.
or string
```

Examples:

```
10 DATA 47.12
100 DATA "ALPHA", "BETA", 22.6, "GAMMA", 74.4```

The DATA statement is used with the READ statement to assign values to variables. The values listed in one or more DATA statements are read sequentially by the READ statement. For example:

```
user: 10 DATA 44.2, 76.4, 18.9 <CR>
20 DATA 100, 47.8, 11.25 <CR>
30 READ A, B, C, D <CR>
40 PRINT "SUM IS "; A+B+C+D <CR>
50 READ X, Y <CR>
60 PRINT "SUM IS "; X+Y <CR>
run <CR>
BASIC: SUM IS 239.5 (44.2 + 76.4 + 18.9 + 100)
SUM IS 59.05 (47.8 + 11.25)
READY
```

TYP() Function

```
General Form:

TYP()                  Returns values 1, 2, or 3, depending on the type of the next DATA
                        item which will be read by the next READ statement.

Value  Type
1      numeric data
2      string data
3      data exhausted
```

Example:

```
10 IF TYP() = 3 THEN 30
20 READ X```

When the TYP() function is encountered the program looks ahead to the next data item in the DATA statement corresponding to the next READ statement. A value of 1, 2 or 3 is returned depending on the type of this data item. The argument 0 must appear. The example above skips a READ statement if the data in the corresponding DATA statement is exhausted. TYP() does not work for file READ.
RESTORE Statement

General forms:

RESTORE

Resets the pointer in the DATA statements so that the next value read will be the first value in the first DATA statement.

RESTORE n

Resets the pointer in the DATA statements so that the next value read will be the first value in the DATA statement at or after line n.

Examples:

10 RESTORE
100 RESTORE 50

The RESTORE statement lets you change the reading sequence in DATA statements. You can start over or move to a particular DATA statement. For example:

user:

10 REA D X, Y, Z <CR>
20 PRINT X+Y+Z <CR>
30 RESTORE 76 <CR>
40 READ X, Y, Z <CR>
50 PRINT X+Y+Z <CR>
60 DATA 100 <CR>
70 DATA 200, 300 <CR>
80 DATA 400 <CR>
RUN <CR>

BASIC: 600

(100 + 200 + 300)

500

(200 + 300 + 400)

END Statement

General form:

END

Terminates execution.

Example:

100 END

The END statement terminates execution of a program, and closes all files that were open. For example:

10 INPUT "WHAT IS THE DIAMETER ", D
20 PRINT "THE CIRCUMFERENCE IS ": 3.1416*D
30 END
40 PRINT "THE AREA IS ": 3.1416*(D/2)^2

When the RUN command is given, only the first three lines of this program are executed. Statement 40 can be executed with the command:

RUN 40 <CR>

4.3. STOPPING OR DELAYING EXECUTION

There are two ways to stop execution of a program from within the program. The END statement ends the execution of a program. The STOP statement stops execution and displays a message telling where execution stopped. The CONT command can be used to resume execution at the next statement. However, any time a program run terminates due to STOP, END, the Mod-Select key, or an error, all open files are closed. The PAUSE statement can be used to delay execution of the following statement for a period of .1 second to 1.82 hours.

ON...RESTORE Statement

General form:

ON exp RESTORE n1, n2, ...

If the value of exp is 1, restores to statement n1, if it is 2, restores to statement n2, etc.

Examples:

10 ON A+3 RESTORE 150
100 ON A RESTORE 200, 300, 350

The ON...RESTORE statement lets you specify the line from which the next data statement will be read. The next READ statement will start reading from the DATA statement selected. For example:

10 READ X, Y, Z, A, B, C
20 ON X+Y RESTORE 100, 110, 120
.
.
100 DATA 4, 1, 0, 4, 7, 2
110 DATA 3, 2, 7, 2, 8, 1
120 DATA 2, 0, 3, 0, 2, 2

Disk BASIC
STOP Statement

**General form:**

```
STOP
```

**Example:**

```
100 STOP
```

The STOP statement stops execution of a program, closes all open files, and displays the message:

```
STOP IN LINE n
```

where n is the line number of the STOP statement. Execution can be continued with the CONT command. For example:

User: LIST <CR>
BASIC: 10 INPUT "WHAT IS THE DIAMETER? ", D
20 PRINT "THE CIRCUMFERENCE IS "; 3.1416*D
30 STOP
40 PRINT "THE AREA IS "; 3.1416*(D/2)^2

User: RUN <CR>
BASIC: WHAT IS THE DIAMETER? 2 <CR> --The user enters 2 for
 THE CIRCUMFERENCE IS 6.2832 the diameter.
 STOP IN LINE 30
User: CONT <CR>
BASIC: THE AREA IS 3.1416 --The CONT command con-
tinues execution with the
 next statement.

PAUSE Statement

**General Form:**

```
PAUSE nexpr
```

Causes a pause before execution of the following statement of duration nexpr tenths of seconds. nexpr may be from 1 to 65535.

**Example:**

```
PAUSE 100
```

Gives a pause of 10 seconds.

The argument nexpr is first evaluated, and truncated to a positive integer between 1 and 65535. A pause of of approximately nexpr tenths of seconds then occurs before the next statement in the program is executed. If nexpr has a value less than 1, it will be truncated to zero and no pause will occur. If nexpr has a value greater or equal to 65536 an error message will appear. The precise duration of the pause is controlled by the clock rate of the microprocessor. In a Sol Terminal Computer with the standard 2.845 MHz jumper installed, the delays will be approximately as given above. If the clock rate is faster or slower, the pause will be proportionately shorter or longer. The maximum delay is 65535 tenths of seconds, or approximately 1.82 hours. Of course multiple PAUSE statements or a loop can create a pause of any length.

4.4. EXECUTION CONTROL

The statements described in this unit allow you to control the order in which statements are executed. With the GO TO and ON...GO TO statements you can branch to a different part of the program. The FOR and NEXT statements let you repeatedly execute a set of statements a specified number of times.

GO TO Statement

**General forms:**

```
GO TO n
GOTO n
```

Transfers control to statement number n.

**Example:**

```
10 GO TO 150
```

The GO TO statement causes a specified statement to be the next statement executed. The statement number can be either greater than or less than the number of the GO TO statement. For example:

```
10 PRINT "ENTER A VALUE FOR X"
20 INPUT X
30 PRINT "X SQUARED IS "; X^2
40 GO TO 10
```

When executed, this program repeats statements 10 through 40 over and over. To escape such an infinite loop, strike the MODE key. For example:

User: RUN <CR>
BASIC: ENTER A VALUE FOR X
User: 216 <CR>
BASIC: X SQUARED IS 100
User: ENTER A VALUE FOR X
User: 55 <CR>
BASIC: X SQUARED IS 25
   ENTER A VALUE FOR X
   7 (The user strikes the MODE key)
STOP IN LINE 20

4-10 Disk BASIC

4-11 Disk BASIC
The ON...GO TO statement lets you branch to one of several statement numbers depending on the value of an expression. If the value of the expression is not an integer, BASIC truncates it to an integer. If there is no statement number corresponding to the value of the expression or truncated expression, the next line is executed.

For example:

**User:** LIST (CR)

**BASIC:**

```
10 INPUT "ENTER VALUES FOR X AND Y ", X, Y
20 PRINT "TYPE 1 TO ADD AND 2 TO SUBTRACT X FROM Y"
30 INPUT N
40 ON N GOTO 60, 70
50 GOTO 10
60 PRINT "THE SUM IS "; X+Y: GOTO 10
70 PRINT "THE DIFFERENCE IS "; Y-X: GOTO 10
```

**User:** RUN (CR)

**BASIC:**

```
10 PRINT "ENTER VALUES FOR X AND Y "; 723.6, 98.84 (CR)
20 GOTO 10
```

**User:** 2 (CR)

**BASIC:**

```
10 PRINT "THE DIFFERENCE IS "; 74.44
```

The value of a variable specified in a FOR statement can be changed within the loop, affecting the number of times the loop executes.
will be executed. For example:

```
10 FOR I = 1 TO 100 STEP -5
20 PRINT I
30 LET I = 50
40 NEXT I
```

This loop will only be executed once because I is set to its final value during the first pass through the loop.

You can include FOR/NEXT loops within other FOR/NEXT loops provided you do not overlap parts of one loop with another. For example:

```
10 FOR A = 1 TO 3
20 FOR B = A TO 30
30 PRINT A * B
40 NEXT B
50 NEXT A
10 LET Y = 10
20 FOR X = 1 TO Y
30 PRINT X + Y
40 NEXT X
50 NEXT Z
```

Note: A GO TO or ON...GO TO statement should not be used to enter or exit a FOR loop. Doing so may produce a fatal error. Use the EXIT statement, described on the next page, to exit a FOR loop.

**EXIT Statement**

**General form:**

```
EXIT n
```

Transfers control to statement n and terminates the current FOR/NEXT loop.

**Example:**

```
10 EXIT 75
```

The EXIT statement allows escape from a FOR/NEXT loop. It causes the specified statement to be executed next and terminates the current FOR/NEXT loop. Only the current FOR/NEXT loop is terminated; if it is nested in others, they will not be terminated. In this respect the statement functions differently than its counterpart in Extended Cassette BASIC. Here is a sample program using EXIT:

```
100 FOR I = 1 TO N
110 FOR J = 1 TO I
120 C = C + 1
130 IF C > 100 THEN EXIT 300
140 NEXT J
150 IF C > 100 THEN EXIT 300
160 NEXT I
170 END
300 PRINT "MORE THAN 100 ITERATIONS"
```

The ON...EXIT statement lets you escape the current FOR/NEXT loop to a statement determined by the value of an expression. If expr or its truncated value corresponds to a statement number following EXIT, the current FOR/NEXT loop is terminated and control is transferred to that statement. If it does not the ON...EXIT statement is ignored. Note that the value of expr must correspond to the position of a statement number in the list following EXIT, not to the value of the statement number itself.

**Example:**

```
10 ON I = 110, 150
160 ON A = B = C EXIT 300, 320, 130
```

The program above operates as follows: When a value of S is read, it is added to 4 and the result is truncated to an integer. If this integer is +1, the current FOR/NEXT loop is terminated and statement 500 is executed; if the integer is +2, statement 600 is executed; if the integer is +3, statement 700 is executed. If the integer is not +1, +2, or +3, the ON...EXIT statement is ignored.
4.3. EXPRESSION EVALUATION

An expression is any combination of constants, variables, functions, and operators that has a numerical or string value. An expression is evaluated by performing operations on quantities preceding and/or following an operator. These quantities are called operands. Examples of some expressions and their operands and operators are:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Operator</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>+</td>
<td>Y</td>
</tr>
<tr>
<td>A</td>
<td>OR</td>
<td>B</td>
</tr>
<tr>
<td>I</td>
<td>NOT</td>
<td>X</td>
</tr>
</tbody>
</table>

The NOT operator precedes an operand. All other operators join two operands.

When BASIC evaluates an expression, it scans from left to right. It performs higher-order operations first, and the results become operands for lower-order operations. For example:

\[ A - B > C \]

The value of \( A - B \) becomes an operand for the \( > \) operator.

Thus, operators act on expressions. The order of evaluation for all BASIC operators is as follows:

Highest

- (unary negation)
- NOT
- *
- /
- >
- \( > = \)
- \( <= \)
- AND
- OR

Lowest

where operators on the same line have the same order, and are evaluated from left to right.

You can enclose parts of a logical expression in parentheses to change the order of evaluation. Expressions in parentheses are evaluated first. For example:

\[ X^2 + 1 \text{ AND } A > B \text{ OR } C = D \]

The results are numerical.

NOTE: BASIC evaluates \( X^n \) faster than it does \( X^n \). Evaluation of \( X^n \) is about the same speed as \( X^n \). Remember that \((-X)^n \) is not allowed, and that \(-X^n \) is equivalent to \((-X)^n \), since unary negation precedes exponentiation.

4.5.1. Arithmetic Operators

The arithmetic operators act on numerical operands as follows:

- exponentiate
- multiply
- divide
- add
- subtract

The results are numerical.

4.5.2. String Operator

The plus operator can be used to concatenate string constants or variables, or expressions. No blanks are added. For example:

User: PRINT "BAR" + "tok" <CR>

BASIC: BARtok

4.5.3. Relational Operators

A relational operator compares the values of two expressions as follows:

expression 1 relational operator expression 2

The result of a relational operation has a numerical value of 1 or 0 corresponding to a logical value of true or false.

The relational operators are:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
</tbody>
</table>

The following expressions with relational operators are evaluated for \( A_1 = 1 \), \( A_2 = 2 \), \( X = 3 \), and \( Y = 4 \):
4.5.4. Logical Operators

The result of a logical operation has a numerical value of 1 or 0, which corresponds to a logical value of true or false. The logical operators AND and OR join two expressions with the following results:

expression1 AND expression2 True only if both expression1 and expression2 are true; otherwise false.

expression1 OR expression2 False only if expression1 and expression2 are false; otherwise true.

The following expressions are evaluated for A = 1, B = 2, and C = 3:

<table>
<thead>
<tr>
<th>Logical Value</th>
<th>Numerical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C &gt; B AND B &gt; A</td>
<td>True</td>
</tr>
<tr>
<td>C &gt; B AND A = B</td>
<td>False</td>
</tr>
<tr>
<td>C = B AND B = A</td>
<td>False</td>
</tr>
<tr>
<td>C &gt; B OR B &gt; A</td>
<td>True</td>
</tr>
<tr>
<td>C &gt; B OR A = B</td>
<td>True</td>
</tr>
<tr>
<td>A &gt; C OR A = C</td>
<td>False</td>
</tr>
</tbody>
</table>

The logical operator NOT reverses the logical value of the expression it precedes. For example, if A, B, and C have the values shown above, the values of logical expression using the NOT operator are as follows:

<table>
<thead>
<tr>
<th>Logical Value</th>
<th>Numerical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT (C &gt; A)</td>
<td>False</td>
</tr>
<tr>
<td>NOT (A = B)</td>
<td>False</td>
</tr>
<tr>
<td>NOT C</td>
<td>False</td>
</tr>
</tbody>
</table>

(C is true because it has a nonzero value.)

4.5.5. Logical and Relational Operations in Algebraic Computations

The numerical value resulting from a logical or relational operation can be used in algebraic computations as shown in the example that follows.

The program below counts the number of 3's in 100 values read from DATA statements:

```
10 FOR I = 1 TO 100
20 READ A
30 LET X = X + (A = 3) When A = 3, X is increased by 1.
40 NEXT I
50 PRINT "OF 100 VALUES ";X;" WERE THREE'S"
100 DATA 1,5,4,6,7,8,9,2,3,4,5,3,2,6,7,8,9,3
110 DATA 4,6,7,4,6,8,2,3,8,4,6,9,6,0,4,0,3,1,3
```

Some examples of expression evaluations in IF statements are:

**IF A > B THEN**
- A > B has a value of true (1) or false (0).
- A has a value of true (nonzero) or false (0).
- A AND B have a value of true (nonzero) or false (0).
- A AND B are true only if both A and B are nonzero.

**IF A < B > C THEN**
- An expression is evaluated from left to right for operators of the same order. In this example, A < B has a value of true (1) or false (0). That value is then compared to C. (1 or 0) > C is either true (1) or false (0).
- Warning: This is not the way to compare B with A and C. For such a comparison, use the AND operator:

```
IF A < B AND B > C THEN...
```

**IF A = B THEN**
- A = B has a value of true (1) or false (0).
- A = B has a value of true (1) or false (0).
- A AND B = C is either true (1) or false (0).
- Warning: This is not the way to test for the equivalence of A, B, and C. For such a test, use the AND operator:

```
IF A = B AND B = C THEN...
```

**IF A = B + C THEN**
- The arithmetic operation is performed first, giving a value for B + C. Then A is either...
equal to that value (true or 1) or not equal to that value (false or 0).

**IF Statement**

**General forms:**

IF `exp` THEN `n`

If the value of `exp` is true, execute statement `n` next.

exp is a logical expression in all of the forms shown here.

n is a statement number in all of the forms shown here.

IF `exp` THEN `n1` ELSE `n2`

If the value of `exp` is true, execute statement `n1` next; otherwise execute statement `n2` next.

IF `exp` THEN `statement1` : `statement2` : ...

If the value of `exp` is true then execute the specified statement(s).

IF `exp` THEN `statement1` : `statement2` : ...

ELSE `statement3` : `statement4` : ...

If the value of `exp` is true then execute the statement(s) following THEN; otherwise execute the statement(s) following ELSE.

Note: The ELSE must appear on the same line as the IF.

IF `exp` THEN `n` ELSE `statement1` : `statement2` : ...

If the value of `exp` is true, execute statement `n` next; otherwise execute the statement(s) following ELSE.

IF `exp` THEN `statement1` : `statement2` : ...

ELSE `n`:

If the value of `exp` is true then execute the statement(s) following THEN; otherwise execute statement `n` next.

Examples:

10 IF A > B THEN 250
100 IF A=C AND NOT B THEN PRINT "ERROR":GO TO 350
260 IF X1 OR Y2 THEN 750 ELSE 305
360 IF NOT R THEN INPUT "R=":R ELSE 780

The IF statement evaluates a logical expression and then takes action based on its value. A true value causes the statement number or statement(s) following THEN to be executed next. If there is an ELSE clause, a false value for `exp` causes the statement number or statement(s) following ELSE to be executed next. Execution continues with the statement following the IF statement, provided control has not been transferred elsewhere.
SECTION 5

5. ADVANCED BASIC

The statements described in this section make Extended BASIC's more powerful features available for use:

* With subroutines and functions, you can define activities that will be performed when a simple call is made or when a function name is specified.
* By using string functions and statements, you can manipulate character data.
* With dimensioned variables, you can set aside storage to quickly and easily manipulate large volumes of data.
* Using the diskette storage and retrieval commands and statements, you can save data for later use.
* With the formatting capabilities of the PRINT statement, you can control the appearance of numeric output.
* Using time and space constraints in the INPUT statement, you can control the response to an INPUT prompt.
* Through cursor-controlling statements and functions, you can draw on the screen.
* Calling upon commands as statements in a program, you can set system characteristics, leave BASIC, and delete the program.
* With the error control statements, you can predetermine a course of action if an error should occur in a program.

5.1. SUBROUTINES

If you have a particular task that must be performed several times during the execution of a program, you can write a subroutine to perform that task and then simply activate the subroutine at the appropriate time. When a subroutine is called from any point in the program, the statements of the subroutine are executed and then control returns to the statement following the calling statement. Variables are not reset or redefined before or after a subroutine's execution.

In Extended BASIC, subroutines are called by specifying the first statement number of the routine in a GOSUB or ON...GOSUB statement. Control returns to the statement after the calling statement when a RETURN statement is encountered.
GOSUB Statement

General form:

GOSUB n  

Executes the subroutine beginning at statement n.

Example:

10 GOSUB 200

The GOSUB statement causes immediate execution of the subroutine starting at the specified statement number. After the subroutine has been executed control returns to the statement following the GOSUB statement. For example:

100 P = 2000, Y = 5, R = .05
110 GOSUB 200
120 PRINT "THE PRINCIPAL AFTER 5 YEARS IS ": P

200 REM: This subroutine finds the principal after Y years on an R% investment of P dollars.
210 FOR N = 1 TO Y
220 P = P + R*P
230 NEXT N
240 RETURN

Calls to subroutines can be included within a subroutine. Extended BASIC allows any level of nested subroutines. Nested subroutines are executed in the order in which they are entered. For example:

100 GOSUB 200
110 PRINT A

200 FOR I = 1 TO R
210 IF I = R GOSUB 370
220 A = A + X^2
230 NEXT I
240 RETURN

370 INPUT "J=", J

This subroutine is executed before the execution of the subroutine at 200 is complete.

RETURN Statement

General form:

RETURN

Transfers control to the statement following the GOSUB or ON...GOSUB statement that called the subroutine.

Example:

100 RETURN

The RETURN statement causes the exit of a subroutine. When a GOSUB or ON...GOSUB statement transfers control to a set of statements ending with a RETURN statement, the line number of the calling statement is saved and control is returned to that line plus one when the RETURN statement is encountered.

A RETURN statement will terminate as many FOR/NEXT loops as necessary to return to the calling GOSUB statement. RETURN statements can be used at any desired exit point in a subroutine. For example:

10 GOSUB 50

50 X = 300
60 FOR I = 1 TO X
70 PRINT I
90 NEXT I
100 RETURN

10 X = 100
20 FOR I = 1 TO X
30 Y = Y + I
40 NEXT I
50 IF X = 0 THEN RETURN
60 RETURN
200 NEXT I

5-2  

Disk BASIC

5-3  

Disk BASIC
ON...GOSUB Statement

General form:

ON exp GOSUB n1, n2, ...

Executes the subroutine beginning
with statement n1 if the
value of exp is 1, executes the
subroutine beginning with
statement n2 if exp is 2, etc.

Examples:

10 ON X GOSUB 120, 150
100 ON S*A/B GOSUB 500
200 ON N GOSUB 90, 500, 10

The ON...GOSUB evaluates, then truncates the expression (exp).
If the value is 1, the subroutine starting at statement n1 is
executed; if 2, the subroutine starting at statement n2 is execut-
ed; etc. If the truncated value of exp is less than 1 or
greater than the number of statements specified, BASIC executes
the next line. After the subroutine has been executed, control
is transferred to the statement following the ON...GOSUB state-
ment. For example:

5 INPUT "ENTER TWO NUMBERS ": X,Y
10 PRINT "DO YOU WANT TO ADD (1), SUBTRACT (2),"
20 PRINT "MULTIPLY (3), OR DIVIDE (4) THE NUMBERS"
30 INPUT I
40 ON I GOSUB 100,200,300,400
50 PRINT "THE ANSWER IS ": A
60 END
100 A = X+Y
110 RETURN
200 A = X-Y
210 RETURN
300 A = X*Y
310 RETURN
400 A = X/Y
410 RETURN

5.2. FUNCTIONS

Functions are similar to subroutines in that they perform a
task that may be required several times in a program. They
differ in that functions can be used in expressions. After
execution, the function itself has a value. For example:

10 LET A = SQR(176) + B

SQR is the square root function and 176 is its argument. When
statement 10 is executed, BASIC computes the square root of 176
and assigns the value to SQR(176); then B is added and the sum
is assigned to A.

5.2.1. General Mathematical Functions

General form:

ABS(exp)
EXP(exp)
INT(exp)
LOG(exp)
LOG10(exp)
RND(exp)
SQR(exp)
SGN(exp)

The absolute value of exp.
The constant e raised to the power exp.
The integer portion of exp (truncation).
The natural logarithm of exp.
The logarithm base 10 of exp.
A random number between 0 and 1.
The square root of exp (exp must be
positive).
The sign of the value of exp; 1 if
positive, -1 if negative, 0 if zero.

Examples:

10 LET X = EXP(X) - LOG(Y)
100 PRINT "THE ANSWER IS ", INT(A*B)
200 IF ABS(X^2-Y^2) > 10 THEN 250

The use of all these functions in a program is straightforward
except for the RND function. This function behaves as if a
table of random numbers were available, and an entry in the
table was returned. The selection of which entry in the table
is returned depends on the argument:

   Argument  Value returned
   0         Returns the next entry in the table
   -1        Returns the first entry, and resets the
             table pointer to the first entry
   n         Returns the entry following n

Although the random numbers generated are between 0 and 1,
numbers in any range may be obtained with an appropriate ex-
pression. The following line gives random integers between
1 and 99:

30 X = INT(RND(0)*100)
5.2.2. Trigonometric Functions

General forms:

\[
\begin{align*}
\text{SIN}(\text{exp}) & \quad \text{The sine of exp radians.} \\
\text{COS}(\text{exp}) & \quad \text{The cosine of exp radians.} \\
\text{TAN}(\text{exp}) & \quad \text{The tangent of exp radians.} \\
\text{ATN}(\text{exp}) & \quad \text{The arctangent of exp; the answer is in radians.}
\end{align*}
\]

\[\text{exp is a numerical expression in all these functions.}\]

Examples:

10 PRINT "THE SIN OF ";Y;" IS ";SIN(Y)
100 LET R = SIN(A)^2 + COS(A)^2
200 IF ATN(14.7) < 1 THEN 400

5.2.3. User-Defined Functions

You can define your own functions making them available for use in the current program. A function’s value is determined by operations on one or more variables. For example, the definition below determines that any time FNA is specified with two values, it will compute the sum of the squares of those values:

\[10 \text{ DEF FNA}(X,Y) = X^2 + Y^2 \quad (X^2 \text{ and } Y^2 \text{ are used instead of } X^2 \text{ and } Y^2 \text{ because the } \ast \text{ operator is faster than the } + \text{ operator for squaring numbers.})\]

The function defined in statement 10 can be used as follows:

100 A = 50, B = 25
110 PRINT FNA(A,B)

When executed, statement 110 will print 50 squared + 25 squared, or 3125.

The rest of this unit describes in detail how to define and use functions of one or more lines.

---

DEF Statement

General forms:

\[
\begin{align*}
\text{DEF FNvar}(\text{var1}, \text{var2}, \ldots) & = \text{expression} \\
\text{variable} & \quad \text{expression} \\
\text{---} & \quad \text{---}
\end{align*}
\]

Defines a one-line function that evaluates exp based on the values of var1, var2, etc.

\[
\begin{align*}
\text{RETURN} & \quad \text{variable} \\
\text{exp} & \quad \text{expression} \\
\text{---} & \quad \text{---}
\end{align*}
\]

Defines a multi-line function that evaluates exp based on the values of var1, var2, etc.

Examples:

10 DEF FN(A,B,C) = A*B/SIN(C)
140 DEF FNA1(R,S)
150 X=R
160 FOR I = 1 TO R
170 X = X + R*S
180 NEXT I
150 RETURN X
160 FNEND

The variables and expression used to define a single-line or multi-line function can be either numeric or string. However, the variables and expression must agree in type. That is, if you are defining a numeric function, use a numerical variable in the function’s name, and return a numeric value as the value of the expression. The same is true for string functions.

Examples are:

10 DEF FNA1(U) = SIN(U) + COS(U)
180 DEF FN$1(US) = "NON" + US
200 DEF FN2(X$) = VAL(X$(2, 4))

In multi-line function definitions, the value returned is the value of the expression on the same line as the RETURN statement. RETURN statements can be used to exit multi-line function definitions as desired. Each definition must end with a FNEND statement. For example:
100 DEF FN(A,B,X,Y)
110 S = 0
120 FOR I = 1 TO X
130 S = S + X*Y
140 NEXT I
150 IF A > B THEN RETURN S - A  --The value of FN will be S-A.
160 RETURN S-B  --The value of FN will be S-B.
170 FNEND

In the above example, the variable names listed in parentheses after FN in line 100 are called formal parameters. In user-defined functions, all formal parameters are locally defined within the function; if any statement in the function modifies the value of a variable which is also a formal parameter, the value of that variable outside the function will NOT be changed. This is true for numerical variables only, not strings, arrays, or matrices. For example:

1 Q = 40
10 DEF FN1(X, Y, Z)
20 X = X + 1, Q = X + Y, Z = Q/3
25 S = 4
30 RETURN Z
40 FNEND
50 X = 1, Y = 2, Z = 3
60 PRINT FN1(X, Y, Z), X, Y, Z, Q, S
RUN
1 1 2 3 3 4
READY

Note that the values of X, Y, and Z, outside the function were not changed by line 20 which is inside the function. Note also that Q, which was not a formal parameter, was changed by line 20. Variable S, introduced within the function, retains its value outside the function.

FNvar Function Call

General form:

FNvar(var1, var2, ... )  Evaluates a user-defined function.

Examples:

10 PRINT FN1(A,B)
100 A1 = FN1(X1,X2,X3)

The FNvar function call evaluates a user-defined function with the same name and assigns the computed value to itself. For example:

5.3. CHARACTER STRINGS

A character string is simply a sequence of ASCII characters treated as a unit. Extended BASIC performs operations with strings as it does with numbers. The string operations use string constants, string variables, string expressions, and string functions.

5.3.1. String Concatenation

You have encountered string constants earlier in this text. The answer is in the statement below is a string constant:

10 PRINT "THE ANSWER IS " ; X+Y

A string constant is indicated in a program by enclosing the characters of the string in quotation marks. However no quotation marks are used when entering a string value from the terminal. Quotation marks cannot be included as part of a string constant.

The size of a string constant is limited only by its use in the program and the memory available.

Some examples of string constants are:

"JULY 4, 1776"
"Dick's stereo"  A string with no characters
"APT #"  is called the null string.
"

In Extended BASIC all lowercase characters are automatically converted to uppercase except for characters in strings or REM statements. Lowercase characters in strings can be entered from or displayed on terminals having lowercase capability. For example:

INPUT SS This string has UPPER- and lowercase characters.
PRINT SS This string has UPPPER- and lowercase characters.
Teletypes print lowercase characters as their uppercase equivalents. If you have a terminal without lowercase capability, refer to the terminal's user's guide to find out how it treats lowercase characters.

Control characters can be included in a string. They may be entered by pressing the control key and the character simultaneously if the character has no immediate function. Or control characters can be typed as \c where c is the character. When a control character is printed, the symbol for the character is displayed or the character's function is performed if it has a function. For example:

10 PRINT "ALPHA \M\JBETA \M\JGAMMA"

prints the following when executed because the function of control-M is carriage return and the function of control-J is line feed:

ALPHA
BETA
GAMMA

To print a single backslash, use this form: "\\". For a list of symbols and functions of control characters, see Section VII of the Sol Systems Manual.

5.3.2. String Variables

A string variable is a variable that can be assigned a string value. To distinguish it from a numerical variable, its symbol is a single letter followed by a dollar sign or a letter, digit, and then a dollar sign. For example: A$, SS$, CS$, Z$.

A string variable can contain one to ten characters unless its maximum size has been declared as a value larger than 10 in a DIM statement.

The assignment statement assigns values to string variables as it does with numerical variables. For example:

10 LET AS $ = "MISSOURI"
100 }$ = A$
200 RS $ = "BOX #", TS $ = "Address"

5.3.3. String Expressions

String expressions can include string constants, string variables, and any of the string functions described later in this unit. In addition they may include the + operator, which means "concatenate" when used with strings. For example:

PRINT "ARGO" + "NAUT" prints ARGONAUT
SS $ = "REASON"
PRINT SS $ + "ABLE" prints REASONABLE

string expressions are treated like numerical expressions in the LET, INPUT, READ, DATA, and PRINT statements. For example:

5 PRINT "WHAT IS THE SOURCE OF THE DATA"
10 INPUT SS$
20 IF SS$ = "DATA" THEN 70
30 INPUT X$, Y$, Z$
40 PRINT "THE LAST VALUE READ WAS \";Z$
60 END
70 READ X$, Y$, Z$
80 GO TO 40
100 DATA "FIRST", "SECOND", "THIRD"

The treatment of strings in logical expressions differs from that of numbers as follows:

1. Strings can be compared using relational operators only within IF statements.
2. No logical operators are allowed in string expressions.

When strings are compared in an IF statement, they are compared one character at a time, left to right. If two strings are identical up to the end of one of them, the shorter is logically smaller. The characters are compared according to their ASCII representations (see Appendix 4). Examples are:

"ASCII" is greater than "073234"
"ALPHA" is greater than "AL"
"94.28" is greater than "$ and name"

The program below shows how an IF statement can be used to compare string values:

10 INPUT "WHAT RANGE OF NAMES DO YOU WANT? ", AS$, Z$
20 FOR I = 1 TO 35
30 READ SS$
40 IF SS$ < A$ THEN 60 Notice that 48 and 50 cannot
50 IF SS$ <= Z$ THEN PRINT SS$ be combined because logical
60 NEXT I operators are not allowed.

String DIM Statement

General form:

DIM var(n) | integer | Specifies the maximum size of a
| string | string that can be contained in var.
| n is the maximum number of characters

Examples:

10 DIM SS(24)
100 DIM AS(72), BS(55), CS(15)
The DIM statement for strings declares the maximum size of a string variable. The maximum size is specified as an integer between 1 and the amount of memory available.

The actual length of the variable at any time is determined by the size of the string currently assigned to it. If a string value with more characters than allowed by the DIM statement is assigned to a variable, the rightmost characters are truncated. For example:

```
10 DIM $S$(12)
20 LET $S$ = "ALPHA IS THE FIRST SERIES"
30 PRINT $S$
```

When executed, this program prints "ALPHA IS THE", the first 12 characters of the string constant.

**SEARCH Statement**

**General form:**

```
SEARCH expl, exp2, var
```

Searches `exp2` for the first occurrence of `expl` and sets `var` to the number of the position at which it is found or 0 if it is not found.

**Examples:**

```
10 SEARCH "CAT",MS,N
100 SEARCH AS, R$, I
```

The SEARCH Statement evaluates `expl` and looks for that string as all or part of the value of `exp2`. If it is found, its location is given by `var`. For example:

```
10 LET XS = "ANOTHER"
20 LET YS = "THE"
30 SEARCH YS, XS, A
40 PRINT A
```

When executed, this program prints 4 as the value of `A` because `THE` begins at the fourth position of `ANOTHER`.

If `expl` is not found the value of `var` is 0.

---

**FILL Statement**

**General form:**

```
FILL string, string expression
```

Fills the string or substring with a copy of the first character in the string expression.

**Examples:**

```
10 FILL AS, CHR(16)
125 FILL X$$(16, 200), " "
```

The FILL statement fills a string specified by a string variable or a substring specified by a substring function with a series of characters identical to the character specified by the string expression. If the string expression yields a string containing more than one character, only the first is used. The expression must yield at least one character. For example:

```
1 DIM AS(5)
10 FILL AS, "XYZ"
RUN
XXXXX
```

One way of displaying a table or other pattern of characters is to use a string variable which represents one line of output. Appropriate elements of the string are then filled with the characters to be displayed. Elements of the string variable that should not show characters may be Filled with blanks. A blank may be represented as CHR(32) or " ".

The FILL statement may also be used as a command.

---

5.3.4. String Functions

The functions described in this unit deal with characters and character strings. The substring function lets you extract or alter part of a string. The LEN function gives the current length of a character string. The ASC and CHR functions perform conversions between characters and their US ASCII codes. The VAL and STR functions convert numbers to strings and vice versa. Finally, the ERR(0) function gives the last error message to appear.
Substring Function

General forms:

\[ \text{var}(\text{expl, exp2}) \]

\[ \text{substring} \]

\[ \text{string expressions} \]

\[ \text{variable} \]

Extracts characters expl through exp2 of the string contained in var.

\[ \text{var}(@) \]

Extracts characters expl through the last character of var.

Examples:

10 LET SS = X$(2,4)
100 LET AS$(1,3) = "NON"
200 INPUT XS$(Q-7)
300 LET IS = L$ + X$(1,5)

The substring function extracts part of a string allowing that section to be altered or used in expressions. The portion of a string to be extracted is indicated by subscripts between 1 and n, where n is the total number of characters in the string. Expressions may be used which yield a value for the subscripts, provided that the value is greater than 1 and less than the number of characters in the string plus two. Noninteger subscript expressions are truncated to integers.

User:

LET AS = "HORSES" <CR>
PRINT AS$(4, 6) <CR> SES  Characters 4 through the end of the string are extracted.

If the subscripts specify a substring not contained within the string it refers to, an error message appears. For example, statements 20 and 30 below result in errors:

10 LET XS = "TERMINAL"
20 LET YS = XS$(1,9)
30 LET ZS = XS$(7,10)

Substrings can be used to change characters within a larger string as shown in the example below:

User:

100 AS = "abcdefghijklmnopqrstuvwxyz" <CR>
200 AS$(3,5) = "123" <CR>
300 PRINT AS <CR>
RUN <CR>
BASIC: ab123efgh

A string may be treated as if it were like an array of subscripted strings, as shown in the example below:

LEN Function

General form:

\[ \text{LEN}(\text{var}) \]

Finds the number of characters in the string currently contained in var.

Examples:

10 PRINT LEN(SS)
100 IF LEN(XS) > 10 THEN 75

The LEN function supplies the current length of the specified string. The current length is the number of characters assigned to the string, not the dimension of the string. For example:

10 DIM SS$(15)
15 LET SS = "COW"
20 PRINT LEN(SS)$

When executed, this program prints 3, the length of the string COW.
ASC and CHR Functions

General forms:

ASC(exp)  Supplies the USASCII code for the first character in the string expression exp.

CHR(exp)  Supplies the character whose USASCII code is given by exp.

Examples:

10 LET I = ASC("a")
200 IF ASC(X$) = 65 THEN PRINT "A"

The ASC and CHR functions perform conversions between characters and their USASCII equivalents. ASC returns the USASCII code for a character whose value is given by a string expression and CHR returns a character whose USASCII code is given by the value of a numerical expression. A table of USASCII codes is presented in Appendix 4.

VAL and STR Functions

General forms:

VAL(exp)  Supplies the numerical value of the string whose value is given by exp.

STR(exp)  Supplies the string value of the number whose value is given by exp.

Examples:

10 X = I * VAL(J$)
200 PRINT VAL(A$).
300 X$ = A$ + STR(I)

The VAL and STR functions perform conversions between decimal numbers and strings that can be converted to numbers. For example:

when executed, this program adds 33.4 to 76.5 and assigns the value, 109.9, to A. Then the STR function converts A to a string and prints the string "109.9".

The STR function produces a string that represents the result of its argument, based on the current default number printing format set by a PRINT statement. For example:

User: PRINT #1003 <CR>
PRINT STR(100.0) <CR>

BASIC: 100.010 Note the use of the 10 character field

User: PRINT #100<CR>
PRINT STR(99999999)

BASIC: $99,999,999 Note the use of the dollar sign ($) and commas (,) as specified in the first PRINT statement.

The VAL function evaluates the string argument as a number. Evaluation stops on the first character which is not legal in an arithmetic constant as described in Section 2.3.1. For example:

User: PRINT VAL("$99,999,999") This statement will result in an IN error due to the $.
PRINT VAL("99,999,999") Evaluation will stop at the first comma.

BASIC: 99

ERR(Ø) Function

General Form:

ERR(Ø) Returns a string consisting of the last error message.

Example:

10 A$ = ERR(Ø)
20 IF A$(1,2) = "NI" THEN PRINT "DELETED FUNCTION USED!"

The ERR(Ø) function returns a USASCII string constant containing the last error message which appeared on the user's terminal. If the ERRSET statement kept the error message from appearing, then the string contains the error message which would have appeared. The argument Ø must be given. Since error messages can take two forms: "XX ERROR", or "XX ERROR IN LINE 00000",

5-16 Disk BASIC
care must be used in comparing the FRR(0) string to other strings. The first two characters in the error message are suf-
ficient to identify which error has occurred, and may be used in comparisons. In the example above, the error message string is stored in string variable A$, then the first two characters of of A$ are compared with 'NF' (Not Implemented). If there is a match, then a message appears on the terminal telling the user he used a version of BASIC which was initialized without a function which his program used. Similar statements can be used to branch to special routines when certain errors occur.

5.4. DIMENSIONED VARIABLES

You can assign many values to a single variable name by allowing additional space for that variable. Such a group of values is called an array and each individual value is an element of that array. The values can be referred to by using subscripts with the variable name. For example, if A1 is an array with 10 elements, individual elements of A1 can be referred to as follows:

A1(1) refers to the first element.
A1(2) refers to the second element.

A1(10) refers to the last element.

An array can have more than one dimension as in the following two-dimensional, 4 by 3 array:

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>15</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2</td>
<td>7.4</td>
<td>8.6</td>
<td></td>
</tr>
<tr>
<td>11.4</td>
<td>4.0</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>8.4</td>
<td></td>
</tr>
</tbody>
</table>

A two-dimensional array is referred to as a matrix. The elements in the example above are referred to by using two sub-
scripts. For example, if the name of the preceding array is T:

T(1,1) = 10
T(1,2) = 15
T(1,3) = 30
T(2,1) = 8.2
T(4,3) = 8.4

To assign additional space to a variable name so that it can contain an array of values, you must dimension it with the DIM statement. The number of dimensions is determined by the number of subscripts specified in the DIM statement.

General forms:

DIM var(1,2,...) Defines an array with one or more dimensions. The size of the array is (ex1*ex2*...)

DIM var(1,2,...),var(2,3,4,...),...

V
e
ules.

The DIM statement allocates space for an array with the specified variable name. The number of dimensions in the array equals the number of expressions in parentheses following the variable name. The number of elements in the array is the product of the expressions.

Examples:

10 DIM A(100)
140 DIM A(4,5),I(L,M-L),J(2,3,10)
200 DIM X(100),SS(72),V(I,J,K)

Elements of an array are referred to as follows:

var(ex1, ex2, ...)

For example:

10 DIM R(5,5)
20 FOR I = 1 TO 5
30 FOR J = 1 TO 5
40 READ R(I,J)
50 NEXT J
60 NEXT I
70 INPUT "WHICH ELEMENT? ",A,B
80 PRINT R(A,B)
100 DATA 7.2, 8.4, 9.4, 8.6, 7.2
110 DATA 3.4, 3.7, 3.8, 9.5, 7.8
120 DATA 7.7, 2.1, 3.2, 5.4, 5.3, 7.6, 5.3, 6.4, 2.1, 2.8
130 DATA 4.8, 9.7, 8.6, 8.2, 11.4

When executed, this program prints the requested elements as shown below:

User: RUN <CR>
BASIC: WHICH ELEMENT? 2,3 <CR> 3.8
User: RUN <CR>
BASIC: WHICH ELEMENT? 3,2 <CR> 2.1
The amount of storage necessary for a given array is given by:

$$9 \times (\text{dimension1}) \times (\text{dimension2}) \times (\text{dimension3}) \ldots \text{etc.}$$

The amount of storage that can be assigned to a variable is determined by the total storage available to BASIC. The memory limit for BASIC can be changed using the command:

```
SET ML = numeric expression
```

To find out how much free storage you have left at any time, use the `FREE` function, which prints the number of bytes of space left for program and variables. For example:

```plaintext
PRINT FREE(0) <CR>
```

2960

5.5. USING DISKETTE FILES FOR DATA STORAGE

The statements described in this unit allow you to store data on diskette, retrieve it, and perform other manipulations.

A data file is a collection of data items stored on diskette under one file name. The user may create, manipulate, or destroy a file. Structurally, a file consists of a set of uniformly sized blocks of disk space. The block structure is of no interest to the BASIC programmer, except that a file divided into large blocks can be read or written faster than a file of small blocks. For this reason, BASIC allows the programmer to specify a file's block size. There is no limit to the number of blocks in a file, except for diskette capacity.

Data stored in diskette files is more permanent than data stored in variables, arrays, or DATA statements. Once data is placed in a file, it can be changed only by a series of special statements designed to change it. Data stored in variables and arrays disappears if the memory containing it is overwritten or if the system power is turned off or fails. The capacity of diskette files is much greater than the amount of system memory which could be made available for the data.

Data in diskette files can be accessed in three ways: serial access, serial access with spacing, and random access, each progressively more complex. File READ and FILE PRINT statements of all three types are available.

In serial access files, data is read or printed as a sequential list of items. Each PRINT statement prints items on the file where the last READ or PRINT statement left off. To read the file, the file is "rewound" to the beginning, and read item by item until the desired items are found, as if the data were stored on magnetic tape. Serial access with spacing is similar to serial access, except that items may be read forward or backward. It is also possible to skip over items in either direction. Random access files have a fundamentally different structure than serial files, described later in this section.

All programs which use diskette data files must request access to the file ("open" the file) with the FILE statement, before any attempt to read writing takes place. The maximum number of files which can be open at one time is limited. The PDOS CONFIG command can be used to change this limit. Each open file requires a certain amount of PDOS buffer memory, which must be addressed for this use. With the correct configuration and adequate memory, up to 16 files may be open at once. Access to open files may be closed with the CLOSE statement.

Two forms of the FILE statement are described below: one for opening serial access files, and one for opening random access files.

For each open file there is a pointer in the file called the file cursor, which keeps track of where the last access ended. Each open file also has an EOF function which keeps track of the last operation performed on the file.

Statements which print data into diskette files can include format elements, as described in Section 5.6, which do not get printed into the file, but control the format in which the data is printed.

The syntax of most data file statements includes the key word, followed by a series of arguments. The arguments are shown enclosed in braces. When the commas separating such arguments are not enclosed within the braces, they themselves must be included within the command, even if the argument is not included. This is to "hold the argument's place", when other arguments will follow. If commas are included within the braces, they may be omitted along with the argument. However, no commas are needed after the last argument; the statement does not need to end in a comma.

Two forms of the FILE statement and three forms each for PRINT and READ are described below. Actually there is only one highly general form each for FILE, PRINT, and READ statements, but presentation of the general forms would be hard to understand. The PRINT and READ statements can include a non-zero expression for cursor displacement ("spacing"), or a non-zero expression for a random number (in which case the file is a random access file). Since spacing is used in serial access files but not random access files, and record numbers are used in random access but not serial access files, the expression for one or the other must equate to zero. When the syntax descriptions below allow for "an expression which, if present, must equate to zero", this is the reason.

With certain limitations, data and program diskette files created in BASIC may be manipulated from within PDOS, and PDOS may be used to create files for use in BASIC. See Section 3.4.2 for a discussion of this subject, and for information about file names for use in the statements below.
Serial Access FILE Statement

General form:

```plaintext
FILE n; name,[access],[ag],[bs]
```

- **name**: A string literal ("this:file/1" for example) or string variable (A8 for example) which is the file's name.
- **access**: An optional number 1, 2, or 3, which specifies what type of access is requested:
  1. READ only. No subsequent PRINT statements.
  2. PRINT only. No subsequent READ statements.
  3. READ or PRINT statements.

Access has not been specified, type 3 access will be requested.

- **ag**: An optional access granted variable. A value of 1, 2, or 3 will be assigned to the variable by the FILE statement, in accordance with the access requested. If no access variable is used, a comma must be inserted to hold the place. Note that an extra comma must be inserted here (since no record size is specified for Serial Access files.)

- **bs**: An expression which specifies the block size to be used by the file. If no block size is specified, and this statement creates the file, a default block size of 256 bytes will be used. bs must be in the range of 1 to 4095 bytes.

Examples:

- 10 FILE #10; SS, 2,, 1024
- 100 FILE #1-F; "file" + STR(3-F)
- 210 FILE #1; "X", 1, X

This form of the FILE statement must be used prior to any of the following file access statements:

1. Serial File PRINT Statement
2. Serial File PRINT with Spacing Statement
3. Serial File READ Statement
4. Serial File READ with Spacing Statement

The **RE**WIND and CLOSE statements may also be used to manipulate the file after the FILE statement.

The FILE statement opens the file (makes it accessible to BASIC), assigns a file reference number for use in the above file access statements, and requests access for reading, printing, or both. If the named file does not already exist, this statement will create it, if the access requested was 2 or 3. A file of a given name may be opened with more than one FILE statement, for different purposes. Provided that different file reference numbers are assigned.

Here is a description of the arguments of the FILE statement:

- **n**: An expression which equates to a file reference number to be assigned.
- **name**: A string literal ("this:file/1" for example) or string variable (A8 for example) which is the file's name.
- **access**: An optional number 1, 2, or 3, which specifies what type of access is requested:
  1. READ only. No subsequent PRINT statements.
  2. PRINT only. No subsequent READ statements.
  3. READ or PRINT statements.

If access is not specified, type 3 access will be requested.

- **ag**: An optional access granted variable. A value of 1, 2, or 3 will be assigned to the variable by the FILE statement, in accordance with the access requested. If no access variable is used, a comma must be inserted to hold the place. Note that an extra comma must be inserted here (since no record size is specified for Serial Access files.)

- **bs**: An expression which specifies the block size to be used by the file. If no block size is specified, and this statement creates the file, a default block size of 256 bytes will be used. bs must be in the range of 1 to 4095 bytes.

The FILE statement sets the file cursor to the first item in the file, and sets its EOF function to 1. (The EOF function is described at the end of this section.) The number of files open at one time is limited (see Section 5.5). Any open file may be closed with the CLOSE statement. Any termination of the run of a program closes all open files.

A given named file may be opened by more than one FILE statement, provided different file reference numbers are assigned. All PRINT statements on the named file must use the first file reference number assigned. Second and subsequent FILE statements assign the value 1 (READ only) in the ag (access granted variable) which prevents printing.

Commas must be inserted to hold the places of items which are not specified in the command, if there are items to follow. No commas are needed after the last item specified. (See Section 5.5.)
Random Access FILE Statement

General form:

FILE #n; name, [access], [ag], [rs], [bs]

<table>
<thead>
<tr>
<th>file reference number</th>
</tr>
</thead>
<tbody>
<tr>
<td>record size expression</td>
</tr>
<tr>
<td>block size expression</td>
</tr>
<tr>
<td>access granted variable</td>
</tr>
<tr>
<td>access requested (1, 2, or 3)</td>
</tr>
<tr>
<td>String or string variable containing file name</td>
</tr>
</tbody>
</table>

Open or creates a random access file of the given name, assigning a file reference number for later use, and requesting read, write, or read/write access.

Example:

FILE #25; "X", ,, 200

This form of the FILE statement is used to open or create a random access file as opposed to a serial access file. The syntax is similar to the Serial Access FILE Statement above, except that an expression is included which specifies a record size. The EOF function is set to 1, as with the Serial Access FILE statement.

A random access file contains sub-structures called records, each a uniformly sized collection of data. Statements which access a random access file must move sequentially through the file to find or print data, but the various records in a random access file may be accessed directly.

The rs (record size) expression specifies how many characters (bytes) can be stored in each record. BASIC actually uses one extra character for each item (collection of characters) in a record. For example, if 3 items, each containing 30 characters are printed in a record, BASIC will use 93 characters of the record. If no record size is specified, the statement becomes a Serial Access FILE statement, described above.

Every FILE statement used with a random access file must include the rs (record size) argument, and each FILE statement which refers to the same named file must specify an rs expression which yields the same value. BASIC cannot maintain the file structure unless this rule is observed.

The Random File READ and PRINT statements, described later in this section, include an extra argument which specifies which record will be accessed.

Serial File PRINT Statement

General form:

PRINT #n; ele1 [,ele2] [,ele3]...

| file reference number |
| expressions to print or format elements |

The values of the expressions are printed sequentially on the referenced file, starting at the current file cursor position.

Examples:

10 PRINT #0; "the answer is", 2 + SQR(N)
105 PRINT #3; 100, 100 + X, 100 + X * 100
230 PRINT #45; %C10F3, X, Y, A, 4D, Q

A previous FILE statement must have already opened the file; n is the file reference number that was assigned by that FILE statement. ele1, ele2, etc., are general expressions which result in numerical or string values to be printed on the file. ele1, ele2, etc., may also be format elements. (See Section 5.6) The expressions are printed sequentially forward on the file, starting at the current file cursor position. If this statement is the first statement after the opening FILE statement to use the file, the beginning file cursor position will be at the end-of-file. Otherwise, the file cursor will be wherever it was left by the last file READ or file PRINT statement. After a statement of this form, the Serial File READ (without spacing) statement cannot be used on the file. This statement leaves the file cursor positioned at the end of the last data item printed. The EOF function for the file is set to 3 (last was PRINT).

For example:

User:  LIST <CR>
10 FILE #3; "EMP", 2
20 DIM SS(30)
30 PRINT "ENTER EMPLOYEE NAMES AND SS #'S"
40 INPUT SS
50 IF SS = "END" THEN CLOSE #3: END
60 PRINT #3; SS
70 GO TO 40
RUN <CR>

BASIC: ENTER EMPLOYEE NAMES AND SS #'S User:  John Dixon 34338749 <CR>     Alfred Dill 322679494 <CR>
       Periodically there is a pause while data
       is written on the diskette file.
BASIC: READY
Serial File PRINT Statement with Spacing

General form:

```
PRINT #n, [re], d; elel [,ele2]... expression(s) to print
expression for file cursor displacement
expression for record number-if present, must
equate to zero in this form of PRINT
file reference number
```

The file cursor of the referenced file is
displaced by d, and the values of elel, ele2,
etc., are sequentially printed on the file.

Examples:

10 PRINT #3, 0, -5; A;$;"CONST", #18F3, 74.8 + B*C
10 PRINT #1, X-4; X(I); Y(J)

Here is a description of the arguments of the Serial File PRINT
with spacing statement:

n The file reference number assigned when the file was
previously opened in a FILE statement.

re An optional expression for record size, which, if
present, must evaluate to zero. Record size may be
other than zero only if n specifies a random access
file.

d The desired file cursor displacement from its present
position. d may range from -65535 to +65535 inclusive.
A displacement of 1 prints the next item in the file.
A displacement of -1 re-prints the last item accessed.
If the displacement d is zero, the file cursor is not
moved, and the statement functions exactly like the Serial
File PRINT statement (without spacing) above.

ele, General expressions which result in numerical or
ele2, string values to be printed on the file. These
etc. expressions may also be format elements as described
in Section 5.6. One or more expressions may be
present.

This statement is the same as Serial File PRINT described above,
except that the file cursor may be moved before printing.
The file which will be printed on must be already opened by a
FILE statement. If this type of PRINT statement is the first
statement executed on the file, the file cursor will be at the
end-of-file. The displacement d will then move the file cursor
relative to the end of file. Otherwise the file cursor will be

Random File PRINT Statement

General Form:

```
PRINT #n, record [,d]; elel [,ele2]... expression(s) to print
file reference number
expression for cursor displacement
if present, must equate to zero
expression which equate to a record number
```

The file cursor of the referenced random
access file is positioned to the specified
record, and the values of elel, ele2, etc.,
are printed on the file.

Example:

```
PRINT #P, 4; "HELLO HUMAN!", "QUE PASA?"
```

Here is a description of the arguments of the Random File PRINT
statement:

n The file reference number assigned when the file was
previously opened in a FILE statement. That FILE
statement must have defined the file as a random
access file, by the inclusion of the rs argument
which specifies record size.

record An expression which evaluates to a record number in
the file, or zero, where the file cursor will be
placed prior to printing. The expression must not
exceed the total number of records in the file plus
one; the file cursor cannot be positioned beyond the
first nonexistent record. If the expression evaluates
to zero, this statement will function exactly like the
Serial File PRINT statement.

d An expression for cursor displacement. Since this
form of the PRINT statement does not use cursor
displacement, this expression must equate to zero,
if present.
ele1, string values to be printed on the file, or format etc., elements as described in Section 5.6. One or more of either type of element may be present.

The file to be printed on must be a random access file, and it must be opened by a prior FILES statement. The file cursor is positioned to the beginning of the specified record, and the values of ele1, ele2, etc., are printed in the record. The EOF function is set to 2.

The sum of the total length of all expressions to be printed, plus the number of such items, is greater than the record size of the file, a record overflow error message is printed, and the program run is terminated.

If the example PRINT statement above is executed on a file containing three records, then record four will be created, and the listed items will be printed into it.

The Serial File PRINT statement may also be used to print on a random access file. However, the Serial File PRINT with spacing statement may not be used.

Serial File READ Statement

General Form:

READ n; [rn] d; [var1,...,varn] ... (statement1; statement2; ...)

file reference number

variable names which receive the read values

Items from the referenced file are read, beginning at the current file cursor position, and assigned to var1, var2, etc. The optional statement(s) is executed only if an end of file is encountered.

Examples:

10 READ #2; A, Y, Z, A, B
100 READ #1; S(1) : PRINT "EOF"; EXIT 200

A FILES statement must have previously opened the file with type 1 or type 3 access. The READ statement reads items, starting at the current file cursor position, and assigns them as the values of the variables. One or more variables may be present. The number of values read is equal to the number of variables present in the statement. If this is the first statement which accessed the referenced file after the FILES statement which opened it, reading will begin at first element of the file. Otherwise, reading will begin from where the file cursor was left by the last access. The statement itself leaves the file cursor positioned just after the last data item read. The EOF function is set to 2.
This statement is the same as Serial File READ (without spacing) except that the file cursor may be moved before reading. A FILE statement must have previously opened the file with type 1 or type 3 access. The file cursor is displaced by d items and enough items are read to fill the variables given. If this type of READ statement is the first statement executed on the file, after the FILE statement, reading will begin with the first item in the file, or the displacement d will move the file cursor relative to the first item. Otherwise the file cursor will be wherever it was left by the last access. This statement itself leaves the file cursor positioned just after the last item read. The EOF function is set to 18.

**Random File READ Statement**

General form:

```
READ #n, rn, d; var1 [,var2]...[statement1; statement2...]
```

- **#n**: file reference number
- **rn**: variable name(s) which receive values
- **d**: expression for cursor displacement. If present, must equate to zero
- **expression which equates to a record number**

The file cursor of the referenced random access file is positioned to the specified record, and items are read into the variables. The optional statement list is executed only if an end of file is encountered.

Examples:

```
10 READ #Q, R9, R; X, Y, Z  :PRINT "EOF" :END
120 READ #F, FNA(X); R9, R8, L4, F
```

Here is a description of the arguments:

- **n**: file reference number assigned when the file was previously opened in a FILE statement. The file must be open with type 1 or 3 access, the the FILE statement must have defined the file as a random access file, by the inclusion of the rs argument that specifies record size.

- **rn**: An expression which evaluates to a record number in the file, or zero, where the file cursor will be placed prior to reading. The expression must not exceed the total number of records in the file plus one; the file cursor cannot be positioned beyond the end-of-file mark. If the expression evaluates to zero, this statement will function exactly like the Serial File READ statement.
REPEAT Statement

General form:
REPEAT n1,n2,...

Repeats the specified number of times.

The REPEAT statement positions the file cursors of the referenced files to the first data item in the files.

If the EOF function for a file is 3 at the time of the CLOSE, the CLOSE statement will end-file the file at the current cursor position. This means that all of the data items after the file cursor are "erased", and the space they occupied will be made available for other files.

CLOSE Statement

General form:
CLOSE n1,n2,...

Closes the specified files.

The CLOSE statement makes the specified files unavailable for reading or writing. They cannot be accessed again until another FILE statement requests access. For example:

110 FILE #1: "NAMES", 2
120 PRINT #1; NS
Here file #1 refers to a file called NAMES.
300 CLOSE #1
210 FILE #1; "SALs", 2
Here file #1 refers to a file called SALS.

File PURGE Statement

General form:
PURGE string [i, var]

A string expression which equates to a file name

The named file is KILLED. If PTDOS gives an error code, it is placed in the variable.

SET XI Statement

General form:
SET XI = n

A variable to accept error code

The file whose name is defined by the string expression is KILLED as described in Section 1.5 of the PTDOS manual. If PTDOS returns an error code, its value is placed in the variable.

The index block (See PTDOS Manual for explanation) of the referenced file is loaded into memory. This allows PTDOS to access the index block faster than from diskette, and therefore speeds execution of Random File READ and PRINT statements.

THIS STATEMENT CANNOT BE USED AS A COMMAND.
**SET FB Command**

**General form:**

```
SET FB = expression
```

- If the expression is zero, static file buffering will be used. If non-zero, dynamic file buffering will be used.

**Examples:**

```
SET FB = 0
SET FB = SGN (200 - RND (0) * 400 / X)
```

See the PTDOS manual for a discussion of buffering. This command can be used as a statement.

**SET OF Command**

**General forms:**

```
SET OF = string  
```

- String expression for file name
  - Sets the output file. Instead of appearing on the terminal or normal output device, all output will be sent to the named file.

```
SET OF = #exp  
```

- Numeric expression for file number
  - Sets the output file to the open PTDOS file whose number is the value of exp.

**Examples:**

```
SET OF = "SLA112"
AS = "STAT12" ; SET OF = AS
SET OF = #2
SET OF = #(N + 3)
```

This statement diverts all output from BASIC, including everything which would normally appear on the video display to the named file. If the named file is a device file (See PTDOS manual), the output will appear on the related device. In the second form, the file exp must be already open. If exp equates to zero, BASIC’s video display driver will be used for output. The # before exp must be contained in the statement, and not supplied by the expression itself. For example:

```
SET OF = #(N + 1) is correct.
AS = "42": SET OF = AS is incorrect.
```

This command may also be used as a statement.

**EOF Function**

**General form:**

```
EOF(file number) Supplies the status of the specified file.
```

- Numerical expression for a file reference number assigned in a prior FILE statement.

**Examples:**

```
10 PRINT EOF(2)
100 IF EOF(1) = 4 THEN 150
```

Every diskette data file which has been opened with a FILE statement has an associated End-Of-File (EOF) function. The EOF function supplies the current status of the specified file as follows:

**VALUE OF EOF**

<table>
<thead>
<tr>
<th>EOF</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>File number was not assigned</td>
</tr>
<tr>
<td>1</td>
<td>Last operation was FILE</td>
</tr>
<tr>
<td>2</td>
<td>Last operation was READ</td>
</tr>
<tr>
<td>3</td>
<td>Last operation was PRINT</td>
</tr>
<tr>
<td>4</td>
<td>Last operation was REWIND</td>
</tr>
<tr>
<td>5</td>
<td>Last operation was READ EOF</td>
</tr>
<tr>
<td>6</td>
<td>Last operation was READ EOF</td>
</tr>
<tr>
<td>8</td>
<td>Last operation was Serial File READ with Spacing</td>
</tr>
<tr>
<td>9</td>
<td>Last operation was Serial File Print with Spacing</td>
</tr>
<tr>
<td>10</td>
<td>Last operation was Random File READ</td>
</tr>
<tr>
<td>11</td>
<td>Last operation was Random File Print</td>
</tr>
</tbody>
</table>
| 12  | Last operation was Random File READ |}

In BASIC you can control the relays in the Sol Terminal Computer or on a CUPS module which are normally used to control the drive motors in cassette recorders. These relays may be used to start and stop audio tapes, or for controlling other equipment. The TUOFF and TUON commands turn the relays off and on. Their forms are:

```
TUOFF
TUON exp  
```

- Numerical expression

The TUOFF and TUON commands can also be used as statements in a program. Their forms as statements is given below.

5-35 Disk BASIC
TUOFF and TUON Statements

General forms:

TUOFF

TUON exp

Turns off both tape motor control units.
Turns on tape motor control unit
exp. exp must evaluate as 1 or 2.

expressions

Examples:

10 TUON 1
100 TUOFF
200 TUON K-1

The TUOFF and TUON statements let you turn on and off the cas- ette recorder motors. They actually control two reed relays, which have isolated low-power contacts appearing at J8 and J9 of the Sol Terminal Computer, or J1 and J2 of the CUTS module. The closure of these contacts under program control can be used for general purpose control applications, provided that
that extra power handling circuitry using relays or semiconduc-
ors is used when necessary. The reed relay contacts are SPST
and will handle .5 Amp, 180 VDC, with a maximum of 18 watts
for a resistive load.

For example:

10 DIM A(100)
20 TUON 1
30 FOR I = 1 TO 100
40 READ A(I)
50 IF A(I) = 0 THEN TUOFF: END
60 NEXT I
70 DATA 1,2,3,4,5,6,7,8,9,0

5.6 CONTROLLING THE FORMAT OF NUMERIC OUTPUT

This section gives additional material about the PRINT
statement which prints on the user's terminal or standard output
device. Forms of the PRINT statement which print on diskette
files are covered in the preceding section, but format elements,
as described in this section, may be included in file PRINT
statements.

In Section 4.1 the PRINT statement was described in its simplest
form, in which the output is automatically formatted. Addi-
tional format specifiers may be added to the PRINT statement
which give great control over the format.

Formatted PRINT Statement

General Form:

PRINT exp, exp,... [format element, exp, exp,...

expressions not
affected by the
format element

expressions
affected by the
format element

Or more generally:

PRINT ele, ele, ele; ele;

commas or semicolons may separate
elements

elements consisting of:
numeric expressions,
string expressions, format elements,
or TAB functions.

Examples:

10 PRINT A; 481; SQR(2 + C); 1#10F3
20 PRINT %5F3; ((A=12) AND B), %D, A, B,
30 PRINT %; A(1,1); "next is"; B(2,2)

The general form consists of zero or more expressions to be
printed according to default format, followed by a format
element, followed by one or more expressions to be printed ac-
cording to the format specified in the format element. The same
PRINT statement can also contain additional format elements
which control additional expressions which follow them. The for-
mat element produces no printed results of its own; it controls
the form in which subsequent numbers are printed. A format ele-
ment controls only the expressions following in the same PRINT
statement, up to the next format element, if any. Using a
special format option it is possible to redefine the default
format used in all following PRINT statements which contain
expressions not controlled by a specific format element.

A format element has the general form:

@[format options]|[format specifier]

The percent sign '%' is required, and distinguishes the format ele-
ment from an expression to be printed. Format options, which
are not required, add special features such as commas, and de-
define the default format. The format specifier, also not re-
quired, defines:

1) The number of columns to be occupied by a PRINTed expres-
sion (field width),
2) The type of number to be printed: integer, floating point, or exponential, and
3) The number of places to the right of the decimal point to be printed.

The following format options are available:

Option  Purpose

S  Places a dollar sign in front of the number
C  Places commas (,) every three places as required, for example: 3,456,789.00
Z  Suppresses trailing zeros after the decimal point.
+  Places a plus sign + in front of all positive numbers. (A minus sign - is always printed in front of negative numbers.)
$  Sets the format element containing it as a new default format used by subsequent PRINT statements, as well as by expressions immediately following.
D  Resets the format to the current default.

Only one format specifier may appear in a format element. Format specifiers have the following four forms:

Specifier  Format

nI  Integer. Numbers will be printed in a field of width n. n must be between 1 and 26. If the value to be printed is not an integer, an error message will be printed.

nFm  Floating Point. Numbers will be printed in a field of width n, with m digits to the right of the decimal point. n must be between 1 and 26, and m must be between 1 and n. Trailing zeros are printed to fill width m, unless the Z option is specified. If the specified field can not hold all the digits in the value to be printed, the value is rounded up to fit.

nEm  Exponential. Numbers will be printed in a field of width n, with m digits to the right of the decimal point. At the end of the field five characters will be printed containing the letter E, a plus or minus sign, and space for an exponent of one to three digits. The exponent may range from -126 to +126. One digit is printed to the left of the decimal point. The field width n must be at least 7 to contain one significant digit plus the 5 characters of the exponential notation. n must be from 7 to 26, and m must be from 0 to n. Here is an example of a number printed in 10E3 format: 1.2345E123. If the specified field can not hold all the digits in the value to be printed, the value is rounded up to fit.

none  Free Format. If a format element consisting of a percent sign alone is used, the format will become the free format as used in the simple unformatted PRINT statement. In free format, integer, floating point, or exponential format is automatically selected depending on the value of the number to be printed, and a field width sufficient to hold all the digits of the number is used. The format options may be added to free format, by using a percent sign followed by one or more format options, with no format specifier.

The field width n in the format specifiers above must be large enough to hold all the characters to be printed, including signs, decimal points, commas, dollar signs, and exponents. If the field width is larger than necessary to contain all the characters to be printed, extra blank spaces are added to the left of the printed characters to fill the field. (In exponential format, blanks are added between the number and its exponent.) Extra field width can be used to create columns of printed output spaced at desired intervals. If semicolons are used to separate the format elements and expressions in a PRINT statement, the field widths given in the format specifiers will be adjoining in the output. This does not mean that numbers printed will have no spaces in between; that depends on whether the number fills its field.

If commas are used to separate the format elements and expressions, there may be extra space added between the fields. The total width of the output is tabulated at fixed 14-character intervals. If a given number has not used the full 14 characters, the field for the next number will begin at the next 14-character interval. In other words, if field widths of 14 or less are used, the numbers will appear in 14-character columns. If field widths of 15 to 24 are used, the numbers will appear in 20-character columns. A mixture of semicolon and comma separators may be used to give variable spacing.

Normally, after a PRINT statement has been executed, the cursor or print head moves to the beginning of the next line, so that the output from the next PRINT statement appears on a new line. If a semicolon is used at the end of a PRINT statement, the return of the cursor or print head is inhibited so that the output from the next PRINT statement will appear on the same line. If a comma is used at the end, the cursor or print head
advances to the beginning of the next 14-character interval, as when commas separate elements within the PRINT statement.

Here are some examples of useful format elements:

**MONETARY FORM:**

\[
\%$C11.2F2
\]

- Floating point form, eleven characters in width, with two of those characters to the right of the decimal point
- Commas will separate every three digits
- Dollar signs will be printed in front of each number

Examples of output:

\[
\begin{align*}
$200.00 & \quad \text{\$9,983.00} \\
35.34 & \quad \text{\$100,000.00}
\end{align*}
\]

**SCIENTIFIC FORM:**

\[
\%E15E7
\]

- Exponential notation, fifteen characters in width, with seven of those characters to the right of the decimal point
- Trailing zeros will be suppressed

Examples of output:

\[
\begin{align*}
1.1414 & \quad \text{E+2} \\
9.40156878-184 & \quad \text{3. E+0}
\end{align*}
\]

The sample program segment below illustrates how format elements can interact:

```
10 PRINT \%$C11.2F2; This statement sets the monetary form given above as the new default format.
20 PRINT A, 42.3, P/I

The values of these expressions will be printed according to the default format in statement 10.
30 PRINT B9; \%+6F8; P, I; \%D; P/I

B9 will be printed according to statement 10. \%+6F8 sets a new format for P and I which follow it. \%D resets the format to the default of statement 10. P/I is printed accordingly.
```

5.7. **CONTROLLED INPUT**

You can include parameters in the INPUT statement to control the number of characters that can be entered from the terminal and the time allowed to enter them. This feature is useful when you want only certain types of answers to questions, or when testing someone's ability to answer quickly.

**Controlled INPUT Statement**

<table>
<thead>
<tr>
<th>General forms:</th>
<th>Controlled INPUT Statement</th>
</tr>
</thead>
</table>
| \( \text{INPUT}(\), \text{char}_1, \text{char}_2, \ldots \) \text{ var1, var2, \ldots } \text{ Enters values from the terminal and assigns them to var1, var2, etc.; however, only char}
| numerical expressions \text{ characters can typed by the user and the user has t tenths of a second to respond.} |

\[
\text{INPUT}(\), \text{char}_1, \text{char}_2, \ldots \text{ message\textquoteright{}\textquoteright{} var1, var2, \ldots } \text{ Same as above, but a message is printed as a prompt before timing begins.}
\]

Examples:

\[
\begin{align*}
10 \text{ INPUT (3,18) X} \\
100 \text{ INPUT (20, 0) N$, AS} \\
200 \text{ INPUT (0,100) A, B, } C \\
300 \text{ INPUT (10,300) "WHAT IS THE DATE?",D$)
\end{align*}
\]

The controlled INPUT statement lets you specify how many characters can be entered and how much time is allowed for response. As soon as \text{char} characters have been typed, BASIC generates a carriage return and accepts no more characters. If the user takes more than \text{t} tenths of a second to respond, BASIC assumes a carriage return was typed. If the optional comma follows INPUT the cursor will remain where the user left it after typing his response, instead of moving to a new line.

If the value of \text{char} is 0, as many as 131 characters can be entered. If the value of \text{t} is 0, the user has an infinite amount of time to respond. For example:

\[
\begin{align*}
5 \text{ DIM AS}(3) \\
10 \text{ FOR X = 1 TO 9} \\
20 \text{ FOR Y = 1 TO 9} \\
30 \text{ PRINT X; \text{" "; Y; \text{" = ";}} \\
40 \text{ INPUT [3, 18H] AS$} \\
42 \text{ IF AS$ = \"\text{"HE WERE SURE SLOW!\"; THEN PRINT \"YOU ARE SURE SLOW!\"; GO TO 30} \\
45 \text{ A = VAL(AS$) \\
50 \text{ IF A < 0 X*Y THEN PRINT \"TRY AGAIN\" \; GO TO 30} \\
60 \text{ NEXT Y} \\
70 \text{ NEXT X}
\end{align*}
\]

5-40 Disk BASIC

5-41 Disk BASIC
When executed, this program accepts a three-character answer from the user and waits 10 seconds for a response. If the user does not respond within 10 seconds the message YOU ARE SURE SLOW is printed. If the user types the wrong response, the message TRY AGAIN is printed.

5.8. ERROR CONTROL

BASIC detects many kinds of errors. Normally, if an error occurs, BASIC will print one of the error messages listed in Appendix 3. However, using the error-control statements described below, you can tell BASIC to execute another statement in the program instead. The ERR(0) function gives a string containing the last error message provided by BASIC, and SYST(0) gives the number of the last PDOS error.

ERRSET and ERRCLR Statements

General forms:

**ERRSET**

Determines that statement n will be executed if any error is detected by number BASIC or PDOS.

**ERRCLR**

Cancels the last ERRSET statement.

Examples:

10 ERRSET 75
160 ERRCLR

The ERRSET n statement lets you determine that statement n will be executed when any error occurs. The error could be an error that would normally result in one of the error messages listed in Appendix 3, or a PDOS error, as listed in the PDOS manual. If an error does occur and the ERRSET n statement does cause a transfer to statement n, before statement n is executed the ERRSET statement itself is cancelled (as if an ERRCLR statement were executed.) Also, the transfer to statement n clears all current FOR/NEXT loops, GOSUS$, and user-defined function calls (as if a CLEAR statement were executed.)

The ERRCLR statement cancels the most recent ERRSET statement. If a statement executed after an ERRSET statement produces an error, BASIC will print a standard error message (See Appendix 3), rather than going to statement n. However, if the ERRSET statement is executed again, it will again set the error trap statement n, as if the ERRSET were encountered for the first time.

**ON...ERRSET Statement**

** General form: **

```
ON exp ERRSET n1, n2, ...
```

Establishes which statement will be executed in the event of an error. If exp is 1, statement n1 is selected, if exp is 2, statement n2, etc.

Examples:

```
10 ON I ERRSET 105,250,490
160 ON A=J ERRSET 50, 300
```

The ON...ERRSET allows you to conditionally determine which statement will be executed if an error occurs. Once an error has occurred, the ON...ERRSET statement is no longer in effect, as if an ERRCLR statement had been executed.

**ERR(0) Function**

**General Form:**

```
ERR(0)
```

Returns a string consisting of the last error message from BASIC.

Example:

```
10 AS = ERR(0)
20 IF AS$1,2 = "NI" THEN PRINT "DELETED FUNCTION USED"
```

The ERR(0) function returns a USASCII string constant containing the last error message which appeared on the user's terminal. If the ERRSET statement kept the error message from appearing, then the string contains the error message which would have appeared. The argument 0 must be given. Since error messages can take two forms: "XX ERROR", or "XX ERROR IN LINE 0000", care must be used in comparing the ERR(0) string to other strings. The first two characters in the error message are sufficient to identify which error has occurred, and may be used in comparisons. In the example above, the error message string is stored in string variable AS$, then the first two characters of AS are compared with "NI" (not implemented). If there is a match, then a message appears on the terminal. Similar statements can be used to branch to special routines when certain errors occur. If the error detected was a PDOS error, ERR(0) will return "FS ERROR". The SYST(0) function below can then be used to determine which PDOS error occurred.
General form:

SYST (exp)

| numeric expression that evaluates to 0

Returns the number of the last PT DOS error.

Example:

10 LET X = SYST(0)
20 PRINT "The last PT DOS error was", SYST(0)

The SYST function returns the number of the last PT DOS error that appeared on the user's terminal. If an ERRSET statement kept the error message from appearing, then SYST(0) returns the number of the message that would have appeared. SYST(0) can be used in conjunction with the ERR(0) function:

10 AS = ERR(0)
20 IF A$(1,2) = "FS" THEN AS = STR(SYST(0))
30 IF AS = "*" THEN PRINT "PROTECTED FILE"

Note that the PT DOS errors, as listed in Section 6 of the first edition of the PT DOS manual, are given as hexadecimal values. SYST(0) returns their decimal equivalents.

5.9. COMMANDS CAN BE STATEMENTS AND STATEMENTS COMMANDS

There are a number of commands that can be included in programs as statements. You have already encountered two: the TUON and TUFF commands. Most commands that can be statements are used for system control. The SET commands set system characteristics and the BYE and SCRATCH commands let you leave BASIC or erase your program. Section 2.5, The Calculator Mode of BASIC, shows how statements may be directly executed without being in a program. Appendix 1, the command and statement summary, lists which commands may be used as statements, and which statements as commands.

5.9.1. The SET Commands

The SET commands let you determine system characteristics. Each SET command except SET ML can be used as a statement in a program. Three SET commands related to diskette data files are covered in Section 5.5. Other SET commands are:

| SET DS = exp1 (exp2) | Sets the video display speed to exp. The larger the value of exp, the slower the display speed. The initial value is 0. If exp2 is present, it determines the availability of the video display speed and spacebar-stop controls during output:
<table>
<thead>
<tr>
<th>exp2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>normal controls</td>
</tr>
<tr>
<td>1</td>
<td>no space bar stopping</td>
</tr>
<tr>
<td>2</td>
<td>no speed control</td>
</tr>
<tr>
<td>3</td>
<td>neither</td>
</tr>
</tbody>
</table>

| SET LL = exp | Sets the output line length to exp. LL is initially set to 64. |
| SET ML = exp | Sets the memory limit. BASIC will not use addresses higher than exp for program or data storage. Cannot be used as a program statement. BASIC initially uses all available memory. |
| SET CP = exp | Sets the character polarity; white characters on black rectangles, or black characters on white. If exp is zero, characters will appear in normal polarity as set by the video display circuitry. If exp is other than zero, characters appear in opposite polarity. Can be used as a program statement. Initially 0. |
| SET CM = exp | Sets the cursor mode. If exp is zero, the cursor will not appear. If exp is other than zero, the cursor will appear. Can be used as a program statement. |

Examples:

User: 10 SET LL = 10 <CR>
20 PRINT "THE LINE IS TOO LONG" <CR>
RUN <CR>
BASIC: THE LINE IS TOO LONG

5.9.2. BYE and SCRATCH Commands

The BYE and SCRATCH commands can be used as statement, so you can exit BASIC from a program or erase the current program. For example:

10 PRINT "NOW I'M HERE"
20 PRINT "NOW I'M NOT"
30 SCRATCH

When executed, this program prints:
and then erases itself.

5.10. CURSOR CONTROL

You can control the position of the cursor or use it to draw on the screen using the CURSOR statement and other devices described in this unit. The current horizontal position of the cursor or print head is given by the POS(0) function.

CURSOR Statement

**General Form:**

CURSOR [exp1],[exp2],[exp3]

Moves the cursor to line exp1 and character exp2. If either is omitted, its value from the last CURSOR statement is used. Exp1 can be any number 1 through 16, and exp1 can be any number 1 through 64. If exp3 is present, the USASCII character whose value is exp3 will be placed at the current cursor position.

**Examples:**

10 CURSOR 1,0,42
100 CURSOR PNA(L)
200 CURSOR X*Y
300 CURSOR Y,45

You can use the CURSOR statement to position the cursor and then use exp3 or a PRINT statement to display a character in that position. You can also print any of the control characters which has an effect on the screen such as \k, which clears the screen. These special characters are described in Section 4.1. The last example prints a hyphen at a position determined by last executed CURSOR statement, since both exp1 and exp2 are absent, and 45 is the decimal USASCII code for hyphen. Appendix 4 contains a table of USASCII codes.

For example:

10 PRINT \k
20 FOR I = .1 TO 3.14 STEP .1
30 LET X = SIN(I)
40 CURSOR I*10,X*10
50 PRINT "*"
60 NEXT I

POS(0) Function

**General Form:**

POS(0)

Return a number between 0 and 131, representing the current horizontal position of the cursor or print head.

**Example:**

10 IF (63 - POS(0)) < LEN(AS) THEN PRINT

In Extended BASIC a line of output from the PRINT statement can be up to 132 characters long. The character positions are numbered 0 to 131 starting from the left. After a PRINT statement and after some other types of operations, the cursor on the video display (or the print head if the output device is a printer or teletype) is left in a new position. The value of the POS(0) function is a number between 0 and 131 representing the current position of the cursor (or print head). If the SET L = exp command or statement has limited the line length to less than 132 characters, the value returned by the POS(0) function will be limited to the new value.

Line length varies with output device. The video display of the Sol Terminal Computer has a line length of 64 characters, but if a line longer than 64 characters is printed, some of the extra characters will be automatically printed on a new line. In the above example the number of characters remaining on the line (63 - POS(0)) is compared with a string AS which will be printed. If the string will not fit on the remainder of the line the statement PRINT is executed which positions the cursor on the beginning of a new line.
SECTION 6

6. MACHINE LEVEL INTERFACE

One of the functions of BASIC is to isolate the user from the operations and requirements of the specific computer on which he is working. BASIC does all interpreting and executing of commands and programs on whatever computer is in use, and the user is free to concentrate only on the logical flow of his program. He can ignore matters such as the absolute locations of his program and data in memory, and the flow of input and output through ports. This isolation could prevent the user from dealing with programs not written in BASIC, and from interfacing with other hardware and software, if special tools were not available within BASIC for doing so.

BASIC provides three tools for addressing absolute memory locations, and three tools for using I/O ports. The POKE statement stores data in a specified memory address, while the PEEK function reads data from a specified address. The CALL function transfers program control to a routine outside of BASIC. The OUT statement places a value in a specified I/O port, while the INP function reads a value from a specified port. The WAIT statement delays program execution until a specified value appears in a port.

Remember that BASIC assumes all numeric expressions are decimal, so all addresses and port numbers must be converted to decimal before use. Appendix 5 contains a table for conversion between hexadecimal and decimal numbers.

In the descriptions of syntax which follow, "numerical expression between 0 and 255" may be interpreted to mean "any expression allowed in BASIC, which, when evaluated, yields a decimal value between 0 and 255."

6-1

Disk BASIC
6.1 WRITING TO A PORT OR MEMORY LOCATION

POKE and OUT Statements

General forms:

POKE exp1, exp2  The value exp2 is stored in memory location exp1.
  - numerical expression between 0 and 255
  - numerical expression from 0 to 65535

OUT exp1, exp2    The value exp2 is sent to I/O port exp2.
  - numerical expression between 0 and 255

Examples:

10 POKE 4095, 11
100 OUT 248, 0

The POKE and OUT statements place a value between 0 and 255 in a specified memory address or I/O port. Since the 8080 microprocessor can address 65,536 memory locations, and has 256 ports, these values are set as limits to the value of exp1. The value of exp2 is converted to a one-byte binary value.

PEEK and INP Functions

General forms:

PEEK(exp)       Supplies the numerical value contained
    in memory location exp
  - numerical expression between 0 and 65535

INP(exp)        Supplies the numerical value contained
    in I/O port exp
  - numerical expression between 0 and 255

Examples:

10 X = PEEK(4095)
100 Y = INP(249)

The PEEK and INP functions return values equal to the contents of memory location or I/O port exp. Since the 8080 processor can address 65,536 memory locations, and has 256 I/O ports, these values are set as limits to the value of exp. One byte is retrieved and its value interpreted as a number between 0 and 255.

Image File LOAD Statement

General form:

LOAD string [, var]
  a string expression which equates to a PTDOS file name

   Loads the named PTDOS image file into memory, and places its starting address in var, if present.

Example:

100 LOAD X$, Y
15 LOAD "GUN"

The LOAD statement loads a PTDOS image file. If var is present, the file's starting address is placed in it. The image file may not be loaded below the "first protected memory address" set upon initialization. The first protected address may be changed with the PTDOS CONFIG command, or the BASIC SET ML command. This statement may be used as a command. However, in a command, "string" must be the actual file name and not a string. The CALL function (below) may be used to execute the loaded image, with the value of var used for expl.

CALL Function

General form:

CALL(exp1[, exp2])  Calls a routine at address exp1, passing optional exp2 in registers D and E, and optionally returning a value in H and L registers.
  - numerical expression between 0 and 65535

Examples:

10 X = CALL(34579)
100 PRINT CALL(18026, 59)
The CALL function invokes a machine language program that begins at address expr1. If expr2 is given, it must be present as a two-byte binary value in the D and E registers of the 8080 when control is transferred. A return address is placed on the 8080 stack, so that a RET or equivalent return instructions at the end of the machine language program may return control to the BASIC program that invoked it. The routine may place a value in the H and L registers to become the value of the CALL function. Since H and L consist of 16 bits, the value returned will consist of a positive integer between 0 and 65535.

**WAIT Statement**

**General Form:**

\[
\text{WAIT expr1, expr2, expr3} \quad \text{Wait until the the value in port expr1 ANDed with expr2, is equal to expr3}
\]

\[
\quad \text{numerical expressions, } 0 \text{ to } 255
\]

\[
\quad \text{numerical expression for port, } 0 \text{ to } 255
\]

**Example:**

\[
\text{WAIT 248, 128, 128}
\]

When a WAIT statement is executed, program execution pauses until a certain value is present in I/O port expr1. To determine this value, expr2, expr3, and the value in port expr1 are converted to one-byte binary values. Each bit in the selected port is "ANDed" with the corresponding bit of expr2. If the result is equal to expr3, program execution continues at the next statement. If the result is not equal to expr3, the program continues to wait for the specified value. Depressing the MODE SELECT key will escape from a WAIT statement.

Expr2 and the logical AND operation provide a way to mask at the selected port bits which are not of current interest. Assume for example that you want a program to wait until bit 7 at port P8 (hexadecimal) becomes 1. (In a SOL, port P8 contains the status of the serial communications channel, and if bit 7 is 1, it means that the UART transmit buffer is empty. The program is going to transmit a character on the serial communications channel, but we need to wait until the UART is empty before placing a new character in the port.*

First look in Appendix 5 and find that the decimal value for P8 is 248, so the first part of the statement is WAIT 248,... Next, create an eight-bit binary mask, with only the bit of interest, bit 7, set to 1: 10000000. Note that a 0 results when a 0 in the mask is ANDed with either 0 or 1 from the selected port. Thus the mask has zeros for all the "don't care" bits. The decimal value for 10000000 binary is 128, so the WAIT statement now consists of WAIT 248, 128,... The value from the port is ANDed with the mask and compared for equivalence with expr3. Since the mask 128 or 10000000 sets the last seven bits of the incoming value from the port to zero, the last seven bits of expr3 must also be zero to achieve a match. You are waiting for bit 7 from the port to become 1. Since you "care" about this bit, bit 7 of the mask is also one, and the result of the AND operation is also one. Thus bit 7 of expr3 should be 1, and the entire byte will be 10000000. Converted to decimal, this value is 128. The complete statement is WAIT 248, 128, 128.

* WAIT cannot be used to monitor the keyboard status port of a SOL Terminal Computer.
SECTION 7

7. MATRIX OPERATIONS

A matrix variable is a numeric variable which has been dimensioned with the DIM statement for two dimensions. A branch of mathematics deals with the manipulation of matrices according to special rules. Extended BASIC contains an extension, described in this section, which allows programs to be written involving matrix calculations according to these special rules. No attempt is made here to present the mathematics of matrices; a prior background is assumed.

Since a matrix has two dimensions, any element is located by two positive integers. One of these integers may be thought of as representing rows and the other columns in a table of values. A three (row) by five (column) matrix arranged as a table and containing real constants is shown below:

```
  | 3.1  4.6  7.0  3.1  0.0 |
-|-|-|-|-|-|
  | 3.1  9.9  0.0  7.2  0.0 |
  | 4.4  1.9  5.6  3.3  0.0 |
```

Before any calculations are made involving matrix variables, the program must first declare the variables to be matrices in a dimension statement. For example:

```
10 DIM A(10, 2), B9(A, B+C),...
```

Here, numeric variable A is given dimensions of 10 rows by 2 columns, and numeric variable B9 is given dimensions of A rows by B+C columns. Any valid BASIC expression may be used as a dimension. Simple variables and matrices of the same name may co-exist in the same program. The matrix A, declared in the example above, is independent of the variable A which has not been dimensioned. Matrix B9 is therefore given a first dimension equal to the value of numeric variable A, not the number of elements in matrix A. In the statement:

```
100 DIM C(5, A(9, 1))
```

matrix C is given 5 rows and a number of columns equal to the value of matrix element A(9, 1). The memory space needed to dimension a matrix is given by the following expression:

```
9 + ((first dimension) * (second dimension) * 6)
```

Since a matrix such as A may co-exist with a variable A in the program, care must be taken to distinguish the two in program statements. In general, A always refers to the variable, while matrix A must have subscripts (A(I, J)).
Matrix elements may be manipulated by all the methods given in earlier sections of this manual. The program below, for example, adds corresponding elements of matrices X and Y into matrix Z:

```
10 DIM X(5, 5), Y(5, 5), Z(5, 5)
20 FOR I = 1 TO 5
30 FOR J = 1 TO 5
40 Z(I, J) = X(I, J) + Y(I, J)
50 NEXT J
60 NEXT I
```

In this respect a matrix can be treated like any multi-dimensional array. This section presents a special group of statements which can manipulate entire matrices in one statement, as compared to the example program above which, while it has the effect of adding two matrices, actually deals with individual matrix elements, one at a time. These special statements all begin with MAT (for matrix). MAT identifies the statement as one dealing with matrices, so within such a statement it is not necessary to include subscripts. For example, the statement

```
10 MAT Z = X + Y
```

accomplishes the same addition process as the program example above, but in only one statement. Note the effect of the same statement without the initial "MAT":

```
10 Z = X + Y
```

Here, the value of X + Y would be assigned to variable Z.

In the descriptions of matrix manipulations which follow, mvar is used to refer to a matrix variable. Shape is used to refer to correspondence in dimensions. The matrix defined by DIM A(5, 2) has the same shape as the matrix defined by DIM B(5, 2), but the matrix defined by DIM C(3, 4) has a different shape. A matrix defined by DIM D(2, 5) is said to have dimensions opposite those of matrices A and B9.

### 7.1. MATRIX INITIALIZATION

The following three statements may be used to define or redefine the contents of a matrix:

- **MAT mvar = ZER** Sets every element in matrix mvar to zero.
- **MAT mvar = CON** Sets every element in matrix mvar to one.
- **MAT mvar = IDN** Sets the matrix to an identity matrix. mvar must have equal dimensions for rows and columns.

### 7.2. MATRIX COPY

If two matrices have the same shape, the values in one may be assigned to the corresponding elements of the other with a statement of the form:

```
MAT mvar1 = mvar2
```

If the matrices in this statement have a different shape, the values will be assigned only where there are corresponding elements with the same subscript. For example:

```
10 DIM A(5, 5), B(10, 2)
20 MAT A = B
```

Here the values in the first five rows of B will be assigned to the five rows of A, but only the first two columns of A will receive new values since B has only two columns. The elements in A which have no corresponding elements in B will retain their original value.

### 7.3. SCALAR OPERATIONS

Each element of a matrix may be added, subtracted, multiplied or divided by an expression and placed into a matrix of the same shape, using a statement of the form shown below:

**Scalar Operations**

```
General Form:
MAT mvar1 = mvar2 op (any expression)
```

Examples:

```
10 MAT A = B * (2.3336)
20 MAT C = D / (2.35 * C(I, J) + SIN(X))
30 MAT E = E + (1)
```

A statement of this form performs the same scalar operation on each element of a matrix. mvar1 and mvar2 must have identical dimensions. The parentheses around expr are required. Matrix elements such as A(5,4) may appear in expr, but not entire matrices. If mvar1 and mvar2 are the same matrix, as in the last example, the resulting new elements will be placed in the old matrix.

---

7-2 Disk BASIC

7-3 Disk BASIC
7.4. MATRIX ARITHMETIC OPERATIONS

A matrix may be added, subtracted, or multiplied (but not divided) by another matrix, and the result placed in a third matrix. A statement of the following general form is used:

\[ \text{MAT} \ m\text{var1} = \ m\text{var1} \text{ op } m\text{var2} \]

_arithmetic operator (+, -, *)_

Differing rules apply, depending on the arithmetic operator used. In addition and subtraction, \( m\text{var1} \), \( m\text{var2} \), and \( m\text{var3} \) must all have the same shape. In multiplication:

1. \( m\text{var3} \) must not be the same matrix as \( m\text{var1} \) or \( m\text{var2} \).
   No check is made to ensure this rule is adhered to.
   If it is broken, unpredictable results will occur.
2. The first dimension (rows) of \( m\text{var3} \) must be the same as the first dimension of \( m\text{var1} \).
3. The second dimension (columns) of \( m\text{var3} \) must be the same as the second dimension of \( m\text{var2} \).
4. The second dimension (columns) of \( m\text{var1} \) must equal the first dimension (rows) of \( m\text{var2} \).

7.5. MATRIX FUNCTIONS

Two matrix functions may be used to place the inverse or transpose of a matrix into another matrix.

**Inverse and Transpose Functions**

<table>
<thead>
<tr>
<th>General Forms:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{MAT} \ m\text{var1} = \text{THN} (m\text{var2})</td>
<td>Places the transpose of ( m\text{var2} ) into ( m\text{var1} ). ( m\text{var1} ) and ( m\text{var2} ) must have opposite dimensions.</td>
</tr>
<tr>
<td>\text{MAT} \ m\text{var1} = \text{INV} (m\text{var2})</td>
<td>Places the inverse of ( m\text{var2} ) into ( m\text{var1} )</td>
</tr>
</tbody>
</table>

**Examples:**

\[ 10 \text{ MAT } A = \text{THN}(B) \]
\[ 20 \text{ MAT } C = \text{INV}(D) \]

\( m\text{var1} \) and \( m\text{var2} \) must not be the same matrix. In both functions, \( m\text{var1} \) and \( m\text{var2} \) must have equal dimensions. No check is made to ensure that \( m\text{var1} \) is not the same matrix as \( m\text{var2} \). If they are the same, unpredictable results will occur. As with all functions, the argument must be within parentheses.

7.6 REDIMENSIONING MATRICES

The total number of elements in a matrix is the product of its two dimensions. In any \text{MAT} statement, a matrix may be given new dimensions, as long as the number of elements is not increased. The new dimensions are assigned merely by giving the new dimensions in parentheses following the matrix variable name. For example:

\[ 10 \text{ DIM } A(20, 20) \]
\[ 20 \text{ MAT } B = A(25, 5) + 1 \]

Here matrix \( A \) is redimensioned from 20 by 20 to 25 by 5, and put in matrix \( B \).

To understand how the elements of the original matrix are reassigned by the new dimensions, consider how the matrix initially dimensioned \text{DIM} \( X(2, 3) \) is reorganized by including new subscripts \( X(3, 2) \). Let us number the original elements:

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
\]

Visualize these same elements in an equivalent linear array (as they are actually stored in the computer's memory):

\[ 1 2 3 4 5 6 \]

When the matrix is given new dimensions, elements are taken row by row from this equivalent linear array. When the last element of the first row is filled, the first element of the second row is filled, and so forth. Here is the resulting arrangement:

\[
\begin{bmatrix}
1 & 2 \\
3 & 4 \\
5 & 6
\end{bmatrix}
\]

If there are more elements in the original matrix than in the new matrix, elements at the end of the equivalent linear array are not assigned to the new matrix, but remain available if another redimension should increase the size. A redimension may only be done in a \text{MAT} statement, and may not be done in a second \text{DIM} statement. The following attempted redimension will not work:

\[ \text{DIM} \ A(10, 10) \]
\[ . . \]
\[ \text{DIM} \ A(5, 5) \]

A matrix variable may appear in a \text{DIM} statement only once. The example above violates this rule.
## APPENDIX 1

Extended BASIC Command and Statement Summary and Index

Minimum keyword abbreviations are underlined. An abbreviation must be followed by a period. Functions and some commands and statements do not have abbreviations. An "S" following a command or statement means it may be also used as a statement; a "C" following a statement means it may be used as a command.

### COMMANDS

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND file --</td>
<td>Reads a program stored on a diskette file and appends it to the current program.</td>
<td>3-13</td>
</tr>
<tr>
<td>BYE</td>
<td>Leaves BASIC and returns to PTDOS. S</td>
<td>2-3</td>
</tr>
<tr>
<td>CAT (/unit)(type)</td>
<td>Displays a catalog of BASIC program or diskette data files, from the specified disk drive unit. of type T, C, S, or R.</td>
<td>3-14</td>
</tr>
<tr>
<td>CLEAR ---</td>
<td>Erases all variable definitions. S</td>
<td>3-9</td>
</tr>
<tr>
<td>CONT --</td>
<td>Continues execution of a program stopped with the MODE key or by a STOP statement.</td>
<td>3-8</td>
</tr>
<tr>
<td>DEL</td>
<td>Deletes all statements.</td>
<td>3-4</td>
</tr>
<tr>
<td>DEL n</td>
<td>Deletes statement n.</td>
<td>3-4</td>
</tr>
<tr>
<td>DEL n1, n2</td>
<td>Deletes statements n1 through n2.</td>
<td>3-4</td>
</tr>
<tr>
<td>DEL n1,</td>
<td>Deletes statements n1 through the last statement.</td>
<td>3-4</td>
</tr>
<tr>
<td>DEL ,n2</td>
<td>Deletes the first statement through statement n2. Note space before comma.</td>
<td>3-4</td>
</tr>
<tr>
<td>EDIT n --</td>
<td>Allows the edit of statement n.</td>
<td>3-6</td>
</tr>
<tr>
<td>GET file --</td>
<td>Reads a diskette file program of type C or T for execution later.</td>
<td>3-12</td>
</tr>
<tr>
<td>KILL file -</td>
<td>Kills the named program file.</td>
<td>3-14</td>
</tr>
<tr>
<td>LIST --</td>
<td>Lists the entire program.</td>
<td>3-3</td>
</tr>
<tr>
<td>LIST n --</td>
<td>Lists statement n.</td>
<td>3-3</td>
</tr>
</tbody>
</table>

APPENDIX 1  A1-1   Disk BASIC
LIST n1, n2
--
Lists statements n1 through n2.

LIST n1,
--
Lists statements n1 through the last statement.

LIST ,n2
--
Lists the first statement through statement n2.

REN
--
Renumber the statements starting with 10 in increments of 10.

REN n
--
Renumber the statements starting with n in increments of 10.

REN n,1
--
Renumber the statements starting with n in increments of 1.

RUN
--
Clears all variable definitions and executes the program beginning with the first line.

RUN n
--
Executes the program beginning with statement n and does not clear variable definitions.

SAVE file [{C},{T}]
--
Saves the current program on a diskette file of the name indicated. C saves the program in semi-compilad format, or T saves the program in text format. The default is C.

SCRATCH
--
Deletes the entire program and clears all variable definitions. S 3-4, 5-45

SET CP=exp
If exp equates to zero, the video cursor will not appear; if exp is non-zero, it will appear. S 3-4, 5-45

SET CP=polarity
If the polarity expression is zero, video characters will appear in normal polarity; if non-zero, characters will appear in reverse video. S 3-4, 5-45

SET DS=speed
Sets the video display speed to the value indicated. S 3-4, 5-45

SET FS=exp
If exp equates to zero, static diskette file buffering will be used; if non-zero, dynamic buffering will be used. 3-4, 5-45

SET L L=length
Sets the line length for BASIC output to the value specified. S 3-4, 5-45

SET ML=size
Sets the memory limit for BASIC to the number of bytes specified. S 3-4, 5-45

SET OF=file
Sets the output file. All output from BASIC is sent to the named file. S 5-34

APPENDIX 1
A1-2
Disk BASIC

SET XI=n
See the statement summary.

TUOFF
Turns off both tape motor relays. S 5-33

TUON unit#
Turns on the specified tape motor relay. S 5-36

XEQ file
Reads and executes a diskette file program of type C or T. 3-10

STATEMENTS

Statement Description Page

CLOSE #file number1, #file number2, ...
Closes the specified files so that they cannot be accessed unless another FILE statement requests access. 5-32

CURSOR [L]{C},{B} Moves the cursor to line L, position C, on the screen. If L or C is omitted, its value from the last CURSOR statement is used. If B is present, the USASCII character whose value is B will be placed at the new cursor position. C 5-46

DATA constant1, constant2, ...
Specifies numerical or string constants that can be read by the READ statement. 3-46

DEF fnvariable(variable, variable2,...) = expression
Defines a one-line function that evaluates an expression based on the values of the variables in parentheses. 4-6

DEF fnvariable(variable, variable2,...)
Defines a multi-line function that executes statements following using the values of the variables in parentheses in calculations, and, when a RETURN statement is encountered, returns the value of the expression on the same line. DEFEND ends the function definition. 5-7

DIM variable(dimension1, dimension2, ...) Defines a multi-dimensional numerical array with the number of dimensions specified. C 5-19

DIM string variable (size)
Declares the number of characters that can be contained in the specified string variable. C 5-11

END
Terminates execution of the program. 4-9

APPENDIX 1 A1-3 Disk BASIC
ERRCLSA
-----
Clears the error trap line number set by the
most recent ERSET statement. C 5-42

ERRSET n --
When an error occurs, BASIC executes statement
next. C 5-42

EXIT n --
Escapes from and terminates the current FOR/
NEXT loop. Statement n is executed next. 4-14

FILE #n#; name, [access], [ag], [rs], [bs]
--
Opens or creates a random access diskette data
file, or if the rs expression is absent or
equates to zero, opens or creates a serial
access file. File reference number n is
assigned to the named file for use in later
statements. An access is requested: 1 for
READ only; 2 for PRINT only, and 3 (default)
for either. If the variable ag is present, it
receives the access granted. If rs is present,
it specifies the record size of a random ac-

cess file. If bs is present, it specifies the
block size (1 to 4095 bytes) to be used. 5-21, 2, 4

FILL string, string expression
Fills a string variable or substring function
with a copy of the first character which the
string expressions yields. C 5-13

FNEND --
Ends a function definition. 5-7

FOR variable = expression1 TO expression2 [STEP interval]
- . The value of expression1 is assigned to the
variable, then the statements between FOR and
NEXT are executed repeatedly until the vari-
able equals expression2. After each iteration
the variable is incremented by 1, or by the
STEP interval if given. 4-13

GOSUB n --
Executes the subroutine beginning at statement
number n. Execution continues with the
statement following the GOSUB statement. 5-2

GOTO n --
Transfers control to statement number n. 4-11

IF expression THEN n
- . -- Executes statement n if the value of the ex-
pression is true; otherwise, executes the next
statement in sequence. C 4-20

IF expression THEN n1 ELSE n2
- . -- Executes statement n1 if the value of the ex-
pression is true; otherwise, executes state-
ment n2. 4-20

INPUT variable1, variable2, ...
-- Accepts values from the terminal and assigns
them to variable1, variable2, etc. C 4-3

INPUT "message", variable1, variable2, ...
-- Displays the message as a prompt and then
accepts values from the terminal, assigning
them to variable1, variable2, etc. C 4-3

INPUT (characters, time) variable1, variable2, ...
-- Accepts values from the terminal and assigns
them to variable1, variable2, etc. The user
can only type the number of characters indi-
cated and has time (in tenths of a second) to
respond. 5-41

INPUT (characters, time) "message", variable1, variable2,...
-- Displays the message as a prompt and then ac-
cepts values from the terminal, assigning them
to variable1, variable2, etc. The user can
only type the number of characters indicated
in parentheses and has time (in tenths of a
second) to respond. 5-41

{LET} variable1 = expression1 [, variable2 = expression2]...
-- Assigns the value of each expression to the
corresponding variable. The word LET may be
absent. C 4-2

APPENDIX 1 A1-4 Disk BASIC

IF expression THEN statement1:statement2:...
- . -- Executes statement1, statement2, etc. if the
value of the expression is true; otherwise,
executes the next statement in sequence. C 4-20

IF expression THEN statement1:statement2:...ELSE statement3:...
- . -- Executes the statements following THEN if the
value of the expression is true; otherwise,
executes the statements following ELSE. C 4-20

IF expression THEN n ELSE statement1:statement2:... 4-20
- . -- Executes statement n if the value of the ex-
pression is true; otherwise, executes the
statements following ELSE.

IF expression THEN statement1:statement2:...ELSE n
- . -- Executes the statements following THEN if the
value of the expression is true; otherwise, executes
statement n. 4-20

APPENDIX 1 A1-5 Disk BASIC
LOAD string [,var]
-- Loads the PT DOS image file, whose name is given by the string expression, into memory. The variable receives its starting address. The file may be executed with the CALL function. C 6-3

MAT mvar = ZE(k)
-- Sets every element in matrix mvar to zero. C 7-2

MAT mvar = CON
-- Sets every element in matrix mvar to one. C 7-2

MAT mvar = IDN
-- Sets the matrix to an identity matrix. C 7-2

MAT mvar1 = mvar2 Copies matrix variable 1 into matrix variable 2. C 7-3

MAT mvar1 = mvar2 op (expr)
-- Performs the same scalar operation on each element of matrix variable 2. op is + or - or + or / C 7-3

MAT mvar3 = mvar1 op mvar2
-- Adds, subtracts, or multiplies matrix variable 1 by matrix variable 2. op is + or - or + or * C 7-4

MAT mvar1 = TRN (mvar2)
-- Places the transpose of matrix variable 2 into matrix variable 1. C 7-4

MAT mvar1 = INV (mvar2)
-- Places the inverse of matrix variable 2 into matrix variable 1. C 7-4

mvar (expression1, expression2)
-- Matrix mvar may be redimensioned by including the new dimensions expression1 and expression2 after the matrix variable name in a MAT statement. C 7-5

NEXT (variable)
-- Ends a FOR loop. C 4-13

ON expression ERSET n1, n2, ...
-- If the value of the expression is 1, sets n1 as the statement to be executed when an error occurs; if the value is 2, sets n2 as the statement to be executed when an error occurs, etc. C 5-43

ON expression EXIT n1, n2, ...
-- If the value of the expression is 1, transfers control to statement n1 and terminates the currently active FOR/NEXT loop; if 2, transfers to statement n2; etc. 4-15

ON expression GOSUB n1, n2, ...
-- If the value of the expression is 1, executes the subroutine starting at statement n1; if the value is 2, executes the subroutine starting at statement n2; etc. 5-4

ON expression GO TO n1, n2, ...
-- If the value of the expression is 1, executes statement n1 next; if it is 2, executes statement n2 next; etc. 4-12

ON expression RESTORE n1, n2, ...
-- If the value of the expression is 1, resets the pointer in the DATA statements so that the next value read is the first data item in line n1; if it is 2, resets the pointer to n2; etc. 4-18

OUT port, value
-- Places the specified value in the indicated I/O port. C 6-2

PAUSE nexpr
-- Delays further execution for nexpr tenths of a second. C 6-10

POKE location, value
-- Places the specified value in the specified memory location. C 6-2

PRINT ele [, ele, ele,...] )
-- Displays numerical or string expression elements according to format elements. Commas or semicolons may separate elements or terminate the PRINT statement. 4-4, 5-37

PRINT #file number; ele [, ele, ele,...]
-- Sequentially prints the values of numerical or string expression elements, according to format elements, onto the referenced diskette data file. C 5-25

PRINT #file number, (record) (), ele (, ele2)...
-- If the file cursor displacement expression d is non-zero, the file cursor is displaced by d and the values of the element(s) are printed on a serial access diskette data file; or, if the record number expression is non-zero, the file cursor is positioned to the given record number in a random access data file, and the values of the element(s) are printed. 5-26, 7
FURGE string {,var}  
---   Kills the diskette data file whose name is  
the value of a string expression. The vari-  
able receives any PT DOS error message.  
5-33

READ variable1, variable2, ...  
---   Reads values from DATA statements and assigns  
them to variable1, variable2, etc.  
4-6

READ n; var1 {,var2}...{:statement1 :statement2}  
---   Reads values from the specified file starting  
at the current file cursor position and  
assigns them to var1, var2, etc. If EOF is  
encountered, the optional statement(s) are  
executed.  
5-28

READ n, {rn}{,d}; var1 {,var2}...{:statement1 :statement2}  
---   If the file cursor displacement expression d  
is non-zero, the file cursor is displaced by d  
and items from a serial access diskette data  
file are read and assigned to var1, var2, etc;  
or, if the record number expression rn is non-  
zero, the file cursor is positioned to the  
given record number in a random access data  
file, and items are read into the variables.  
If EOF is encountered, the optional state-  
ment(s) are executed.  
5-29, 30

REM any series of Characters  
The characters appear in the program as  
remarks. The statement has no effect on  
execution.  
4-1

RESTORE (n)  
---   Resets the pointer in the DATA statements  
to the beginning. If n is present, the  
pointer is set to the first data item in  
statement n.  
4-8

RETURN  
---   Returns from a subroutine.  
5-3

RETURN exp  
---   Returns from a function. The value returned is  
exp.  
5-7

REWIND #file number1, #file number2, ...  
---   Rewinds the specified files.  
5-32

SEARCH string expression1, string expression2, variable  
---   Searches the second string for the first  
occurrence of the first string specified. The  
variable is set equal to the character posi-  
tion at which the first string was found.  
If it is not found, the variable is set equal  
to zero.  
5-12

SET XI=number  
---   Loads the index block of the referenced file  
into memory for fast access.  
5-32

STOP  
---   Terminates execution of the program and prints  
"STOP IN LINE n" where n is the line number of  
the STOP statement.  
4-10

WAIT expl1, expl2, expl3  
---   The next statement is not executed until the  
value in port expl1, ANDed with expl2, is equal  
to expl3.  
6-4

XEQ file  
---   Reads the program from the specified diskette  
file and begins execution. The file name  
is a string expression so it must be enclosed  
in quotation marks if given directly.  
3-13

APPENDIX 1  A1-8  Disk BASIC

APPENDIX 1  A1-9  BASIC
APPENDIX 2
EXTENDED BASIC FUNCTION SUMMARY AND INDEX

In the function forms below, which are arranged alphabetically, n represents a numeric expression and s represents a string expression. Function names may not be abbreviated.

<table>
<thead>
<tr>
<th>Function</th>
<th>Value Returned</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(n)</td>
<td>The absolute value of the numerical expression n.</td>
<td>5-5</td>
</tr>
<tr>
<td>ASC(s)</td>
<td>The USASCII code for the string expression s. Only the first character of the string is interpreted.</td>
<td>5-16</td>
</tr>
<tr>
<td>ATN(n)</td>
<td>The arctangent of the numerical expression n in radians.</td>
<td>5-6</td>
</tr>
<tr>
<td>CALL(address[,parameter])</td>
<td>The value in HL. CALL places a return address on the 8080 stack, calls the routine at the specified memory address, and optionally passes the value of a parameter in the DE register. The routine may return a value in HL, which becomes the value of the CALL function.</td>
<td>6-3</td>
</tr>
<tr>
<td>CHR(n)</td>
<td>The character whose USASCII code is the value of numerical expression n.</td>
<td>5-16</td>
</tr>
<tr>
<td>COS(n)</td>
<td>The cosine of n in radians.</td>
<td>5-6</td>
</tr>
<tr>
<td>EOF(file number)</td>
<td>The status of the specified file. 0 file number was not assigned 1 last operation was FILE 2 last operation was READ 3 last operation was PRINT 4 last operation was REWIND 5 last operation was READ EOF (end of record) 6 last was READ EOF (end of file) 18 last was Serial File READ with Spacing 19 last was Serial File PRINT with Spacing 34 last was Random File READ 35 last was Random File PRINT 37 last was Random File READ EOF 38 last was Random File READ EOF</td>
<td>5-35</td>
</tr>
<tr>
<td>ERR(0)</td>
<td>A string containing the last error message.</td>
<td>5-17, 5-43</td>
</tr>
<tr>
<td>EXP(n)</td>
<td>The constant e raised to the power n.</td>
<td>5-5</td>
</tr>
<tr>
<td>FNvariable/variable1, variable2, ...)</td>
<td>The value of user-defined function FNvariable. variable1, variable2, etc. are arguments.</td>
<td>5-7</td>
</tr>
</tbody>
</table>
The number of bytes of space left available in BASIC for program and variables. 5-18

Supplies the numerical value contained in I/O port exp. Exp is between 0 and 255. 6-2

Truncates n to its integer part. 5-5

The number of character in the string variable whose name is specified. 5-15

The natural logarithm of n. 5-5

The logarithm base 10 of n. 5-5

The value contained in memory location n. 6-2

The current position of the cursor (0-131). 5-47

A random number between 0 and 1. exp = 0, -1 or n. 5-5

The sign of the value of n; 1 if positive, -1 if negative, 0 if n is zero. 5-5

The sine of n in radians. 5-6

The square root of n. 5-5

The character representation of the value of n. 5-16

Return the number of the last PTDOS error, where n evaluates to 0. 5-44

Moves the cursor or print head horizontally to character position n. Use only in a PRINT statement. 4-5

The tangent of n in radians. 5-6

A value representing the type of data that will be read from the DATA statement corresponding to the next READ statement: 1 for numeric data, 2 for string data, or 3 for data exhausted. 4-7

The numerical value of the string s. The value of s must be convertible to a legal numerical constant. 5-16

Characters exp1 through exp2 of the specified string if exp2 is present. Characters exp1 through the end of the specified string if exp2 is omitted. 5-14

An element of an array with the specified name. The element's position is given by n1, n2, etc. 5-16

APPENDIX 3

ERROR MESSAGES

All errors are fatal and stop the execution of the program or command causing the error, unless an ERRSET statement is in effect. If the error occurs while writing data on a file or saving a program, some information may be lost. Errors are arranged below alphabetically by error message.

There are many errors that PTDOS may print while BASIC is running. Such messages have the form: "PTDOS ERROR message" instead of "xx ERROR..." PTDOS errors are listed in Section 6 of the PTDOS manual.

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
<th>What to Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Access error. An attempt has been made to access a file in the wrong mode (read, write, or read/write). Incorrect.</td>
<td></td>
</tr>
<tr>
<td>AM</td>
<td>Argument error. A function has been called with the wrong number or type of arguments.</td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>Cannot append. The file indicated in the last APPEND command is the wrong type. It must be a text format file.</td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td>Can't convert. The last VAL function attempted to determine the value of a string which did not contain a number.</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>Control stack error. Possible causes are: -RETURN without a prior GOSUB -Incorrect FOR/NEXT nesting -Too many nested GOSUBs -Too many nested FOR loops -Too many nested function calls</td>
<td></td>
</tr>
<tr>
<td>DD</td>
<td>Double definition. An attempt has been made to define a function with a name that is already defined.</td>
<td></td>
</tr>
</tbody>
</table>

APPENDIX 3

A3-1

Disk BASIC
DI  Direct execution error. The statement last typed cannot be executed in calculator mode.

DM  Dimension error. A dimension statement contains a variable name that is already dimensioned or cannot be dimensioned.

DZ  Divide by zero error. An expression in the last statement attempted to divide by zero.

PD  Format definition error or file declaration error. The last PRINT statement contained a bad format definition, the last statement referring to a file number specified an undeclared file, or the last FILE statement could not declare the file as requested.

FM  Format error. A field definition in the last formatted PRINT statement is not large enough or it is too large.

FN  File name error. A filename is too short, too long, or contains illegal characters.

FO  Field overflow. An attempt has been made to print a number larger than extended BASIC's numerical field size.

FP  Floating Point error. BASIC cannot handle numbers greater than 10 to the 126th power, or less than 10 to the -126th power.

FS  File Structure error. A PTOS error occurred.

Give the statement a line number and execute it as all or part of a program.

Rename the dimensioned variable. Make sure the variable name is valid.

Set the value of the divisor to a nonzero number before dividing.

Either check the format definition against the documentation under "Formatted PRINT Statement" or find the most recent FILE statement and verify its syntax and the file number declared.

Use the PRINT statement in calculator mode to determine the size of the value to be printed. Adjust the field declaration accordingly.

Check for spelling errors or use a different name.

Display values used to compute the number. Trace the source of the overflow in reverse order through the program.

No solution.

Use the SYST() function to determine the error number.

Input error. The ERRSET statement is in effect and non-numeric input was given to a numeric INPUT statement.

Internal stack error. An expression was too complex to evaluate.

Line too long. The next line to be listed is too long for BASIC. It cannot be edited or saved in the text mode.

Line number reference error. A statement referred to a line that does not exist.

Matrix Dimension Error. Dimensions are incompatible with the operation attempted.

Memory Protect error. An attempt was made to overwrite BASIC or the current BASIC program. This error can be produced by the LOAD command/statement.

Matrix Singular Error. The operation attempted cannot be performed on a singular matrix.

Not available. A command is not presently available—for example commands to the video display driver are not available when output is set to another file.

Rerun the program, using appropriate input.

Divide the expression into parts, using assignment statements.

If you don't know the number of the next line to be listed, renumber the program and give the LIST command again. Replace the long line with shorter lines. You cannot list the long line, so you must reconstruct its meaning from the context of the surrounding statements.

List the area of the program around the line referred to. Find the correct line number and revise the reference.

Redimension the matrix or restructure the operation.

Check image file load address if the LOAD statement was used.

The operation cannot be performed on the data in the given matrix.

Return output to the video driver, using SET DF=8, or don't use the offending command.
NC  Not CONTInuable. The current program, if any, cannot be CONTInued.

NI  Not implemented. An attempt was made to use matrix or trig functions which were deleted.

NP  No program. BASIC was instructed to act on the current program and none exists.

GB  Out of bounds. The argument or parameter given in not within the range of the function or command last executed.

RO  Record overflow. An attempt was made to write more items into a record of a random access data file than the record could hold.

SN  Syntax error. The statement or command last executed was constructed incorrectly.

SO  Storage overflow. There is insufficient storage to complete the last operation.

TY  Type error. The variable or function name appearing in the last statement is the wrong type. The types are string, variable, simple variable, dimensioned variable, and function.

UD  Undimensionned matrix. A variable name was used which was not previously dimensioned.

UR  Unresolved linenumber reference. During a RUNumber command, a control transfer statement referred to a non-existent line number.

Make sure a BASIC program is ready to run. You cannot CONTInue after editing a program, using the CLEAR command, etc.

Display the values of the arguments or parameters used. If they seem reasonable, look up the definition of the function or command.

Write the extra items into a new record, write less items per record, or rewrite the file with a new record size.

Check the syntax of the command or statement in Appendix 1.

Use the FREE command to find out how much storage is left. Use SET ML to change the memory limit for BASIC.

Check the names of functions and dimensioned variables. Make sure the operation is appropriate for the type of data indicated.

DIMension the matrix in an earlier DIM statement.

Look for typos in the program. Unresolved references will have been changed to 8.
## APPENDIX 4

**TABLE OF ASCII CODES (Cont'd) (Zero Parity)**

<table>
<thead>
<tr>
<th>Paper tape</th>
<th>Upper Octal</th>
<th>Octal</th>
<th>Decimal</th>
<th>Hex</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>123. 45678P</td>
<td>1234567890</td>
<td>A900</td>
<td>44</td>
<td>40</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>B</td>
<td>45</td>
<td>A</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>C</td>
<td>46</td>
<td>41</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>D</td>
<td>47</td>
<td>42</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>E</td>
<td>48</td>
<td>43</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>F</td>
<td>49</td>
<td>44</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>G</td>
<td>50</td>
<td>45</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>H</td>
<td>51</td>
<td>46</td>
<td>h</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>I</td>
<td>52</td>
<td>47</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>J</td>
<td>53</td>
<td>48</td>
<td>j</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>K</td>
<td>54</td>
<td>49</td>
<td>k</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>L</td>
<td>55</td>
<td>50</td>
<td>l</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>M</td>
<td>56</td>
<td>51</td>
<td>m</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>N</td>
<td>57</td>
<td>52</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>O</td>
<td>58</td>
<td>53</td>
<td>o</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>P</td>
<td>59</td>
<td>54</td>
<td>p</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>Q</td>
<td>60</td>
<td>55</td>
<td>q</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>R</td>
<td>61</td>
<td>56</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>S</td>
<td>62</td>
<td>57</td>
<td>s</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>T</td>
<td>63</td>
<td>58</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>U</td>
<td>64</td>
<td>59</td>
<td>u</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>V</td>
<td>65</td>
<td>60</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>W</td>
<td>66</td>
<td>61</td>
<td>w</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>X</td>
<td>67</td>
<td>62</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>Y</td>
<td>68</td>
<td>63</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>Z</td>
<td>69</td>
<td>64</td>
<td>z</td>
</tr>
</tbody>
</table>

## APPENDIX 5

**HEXADECIMAL-DECIMAL INTEGER CONVERSION TABLE**

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 000</td>
<td>4 096</td>
<td>20 000</td>
<td>131 072</td>
</tr>
<tr>
<td>02 000</td>
<td>8 192</td>
<td>30 000</td>
<td>196 608</td>
</tr>
<tr>
<td>03 000</td>
<td>12 288</td>
<td>40 000</td>
<td>262 144</td>
</tr>
<tr>
<td>04 000</td>
<td>16 384</td>
<td>50 000</td>
<td>327 680</td>
</tr>
<tr>
<td>05 000</td>
<td>20 480</td>
<td>60 000</td>
<td>393 216</td>
</tr>
<tr>
<td>06 000</td>
<td>24 576</td>
<td>70 000</td>
<td>458 752</td>
</tr>
<tr>
<td>07 000</td>
<td>28 672</td>
<td>80 000</td>
<td>524 288</td>
</tr>
<tr>
<td>08 000</td>
<td>32 768</td>
<td>90 000</td>
<td>589 824</td>
</tr>
<tr>
<td>09 000</td>
<td>36 864</td>
<td>A0 000</td>
<td>655 360</td>
</tr>
<tr>
<td>0A 000</td>
<td>40 960</td>
<td>B0 000</td>
<td>720 896</td>
</tr>
<tr>
<td>0B 000</td>
<td>45 056</td>
<td>C0 000</td>
<td>786 432</td>
</tr>
<tr>
<td>0C 000</td>
<td>49 152</td>
<td>D0 000</td>
<td>851 968</td>
</tr>
<tr>
<td>0D 000</td>
<td>53 248</td>
<td>E0 000</td>
<td>917 504</td>
</tr>
<tr>
<td>0E 000</td>
<td>57 344</td>
<td>F0 000</td>
<td>983 040</td>
</tr>
<tr>
<td>0F 000</td>
<td>61 440</td>
<td>100 000</td>
<td>1 048 576</td>
</tr>
<tr>
<td>10 000</td>
<td>65 536</td>
<td>200 000</td>
<td>2 097 152</td>
</tr>
<tr>
<td>11 000</td>
<td>69 632</td>
<td>300 000</td>
<td>3 145 728</td>
</tr>
<tr>
<td>12 000</td>
<td>73 728</td>
<td>400 000</td>
<td>4 194 304</td>
</tr>
<tr>
<td>13 000</td>
<td>77 824</td>
<td>500 000</td>
<td>5 242 880</td>
</tr>
<tr>
<td>14 000</td>
<td>81 920</td>
<td>600 000</td>
<td>6 291 566</td>
</tr>
<tr>
<td>15 000</td>
<td>86 016</td>
<td>700 000</td>
<td>7 340 333</td>
</tr>
<tr>
<td>16 000</td>
<td>90 112</td>
<td>800 000</td>
<td>8 388 688</td>
</tr>
<tr>
<td>17 000</td>
<td>94 208</td>
<td>900 000</td>
<td>9 437 128</td>
</tr>
<tr>
<td>18 000</td>
<td>98 304</td>
<td>A0 000</td>
<td>10 485 760</td>
</tr>
<tr>
<td>19 000</td>
<td>102 400</td>
<td>B0 000</td>
<td>11 534 356</td>
</tr>
<tr>
<td>1A 000</td>
<td>106 496</td>
<td>C0 000</td>
<td>12 582 912</td>
</tr>
<tr>
<td>1B 000</td>
<td>110 592</td>
<td>D0 000</td>
<td>13 631 488</td>
</tr>
<tr>
<td>1C 000</td>
<td>114 688</td>
<td>E0 000</td>
<td>14 680 064</td>
</tr>
<tr>
<td>1D 000</td>
<td>118 784</td>
<td>F0 000</td>
<td>15 728 640</td>
</tr>
<tr>
<td>1E 000</td>
<td>122 880</td>
<td>1 000 000</td>
<td>16 777 216</td>
</tr>
<tr>
<td>1F 000</td>
<td>126 976</td>
<td>2 000 000</td>
<td>17 836 792</td>
</tr>
</tbody>
</table>

*To convert numbers above those ranges, add table values to the figures below.*

---

APPENDIX 4

**A4-2**

Disk BASIC

APPENDIX 5

**A5-1**

BASIC
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00000001</td>
<td>00020003</td>
<td>00040005</td>
<td>00060007</td>
<td>00080009</td>
</tr>
<tr>
<td>01</td>
<td>00010002</td>
<td>00030004</td>
<td>00050006</td>
<td>00070008</td>
<td>0009000A</td>
</tr>
<tr>
<td>02</td>
<td>00020003</td>
<td>00040005</td>
<td>00060007</td>
<td>00080009</td>
<td>000A0010</td>
</tr>
<tr>
<td>03</td>
<td>00030004</td>
<td>00050006</td>
<td>00070008</td>
<td>0009000A</td>
<td>000B0010</td>
</tr>
<tr>
<td>04</td>
<td>00040005</td>
<td>00060007</td>
<td>00080009</td>
<td>000A0010</td>
<td>000C0012</td>
</tr>
<tr>
<td>05</td>
<td>00100006</td>
<td>00120008</td>
<td>00140010</td>
<td>00160012</td>
<td>00180014</td>
</tr>
<tr>
<td>06</td>
<td>00110007</td>
<td>00130009</td>
<td>00150011</td>
<td>00170013</td>
<td>00190015</td>
</tr>
<tr>
<td>07</td>
<td>00120008</td>
<td>00140010</td>
<td>00160012</td>
<td>00180014</td>
<td>001A0016</td>
</tr>
<tr>
<td>08</td>
<td>00130009</td>
<td>00150011</td>
<td>00170013</td>
<td>00190015</td>
<td>001B0017</td>
</tr>
<tr>
<td>09</td>
<td>00140010</td>
<td>00160012</td>
<td>00180014</td>
<td>001A0016</td>
<td>001C0018</td>
</tr>
<tr>
<td>0A</td>
<td>00150011</td>
<td>00170013</td>
<td>00190015</td>
<td>001B0017</td>
<td>001D0019</td>
</tr>
<tr>
<td>0B</td>
<td>00160012</td>
<td>00180014</td>
<td>001A0016</td>
<td>001C0018</td>
<td>001E0020</td>
</tr>
<tr>
<td>0C</td>
<td>00170013</td>
<td>00190015</td>
<td>001B0017</td>
<td>001D0019</td>
<td>001F0021</td>
</tr>
<tr>
<td>0D</td>
<td>00180014</td>
<td>001A0016</td>
<td>001C0018</td>
<td>001E0020</td>
<td>00200022</td>
</tr>
<tr>
<td>0E</td>
<td>00190015</td>
<td>001B0017</td>
<td>001D0019</td>
<td>001F0021</td>
<td>00220023</td>
</tr>
<tr>
<td>0F</td>
<td>001A0016</td>
<td>001C0018</td>
<td>001E0020</td>
<td>00200022</td>
<td>00240024</td>
</tr>
</tbody>
</table>

**APPENDIX 5**

<table>
<thead>
<tr>
<th>A5-2</th>
<th>BASIC</th>
</tr>
</thead>
</table>

**APPENDIX 5**

| A5-3 | BASIC |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
APPENDIX 6

BIBLIOGRAPHY

1. An Introduction to Microcomputers, Volume 0, The Beginner's Book
   Adam Osborne and Associates, Inc. 1977

2. BASIC Programming in Real Time
   Don Cassel, 1982
   Reston Publishing Company, Inc. 1975

3. Simplified BASIC Programming
   Gerald A. Silver
   McGraw-Hill Co. 1974

4. Basic BASIC
   James S. Coan
   Hayden 1970

5. Advanced BASIC
   James S. Coan
   Hayden 1977

6. Problems for Computer Solution
   Fred Gruenberger and George Jaffray
   Wiley 1965

7. BASIC
   Samuel L. Marateck
   Academic Press 1975

8. Some Common BASIC Programs
   Lon Poole and Mary Borchers
   Adam Osborne & Associates, Inc. 1977

9. Game Playing with BASIC
   Donald D. Spencer
   Hayden Book Company, Inc. 1977

10. Game Playing with Computers
    Donald D. Spencer
    Hayden 1975

11. 101 BASIC Computer Games
    Digital Equipment Corporation 1975

12. Theory and Problems of Matrices
    Schaum's Outline Series
    Frank Ayres, Jr.
    McGraw-Hill Co. 1962