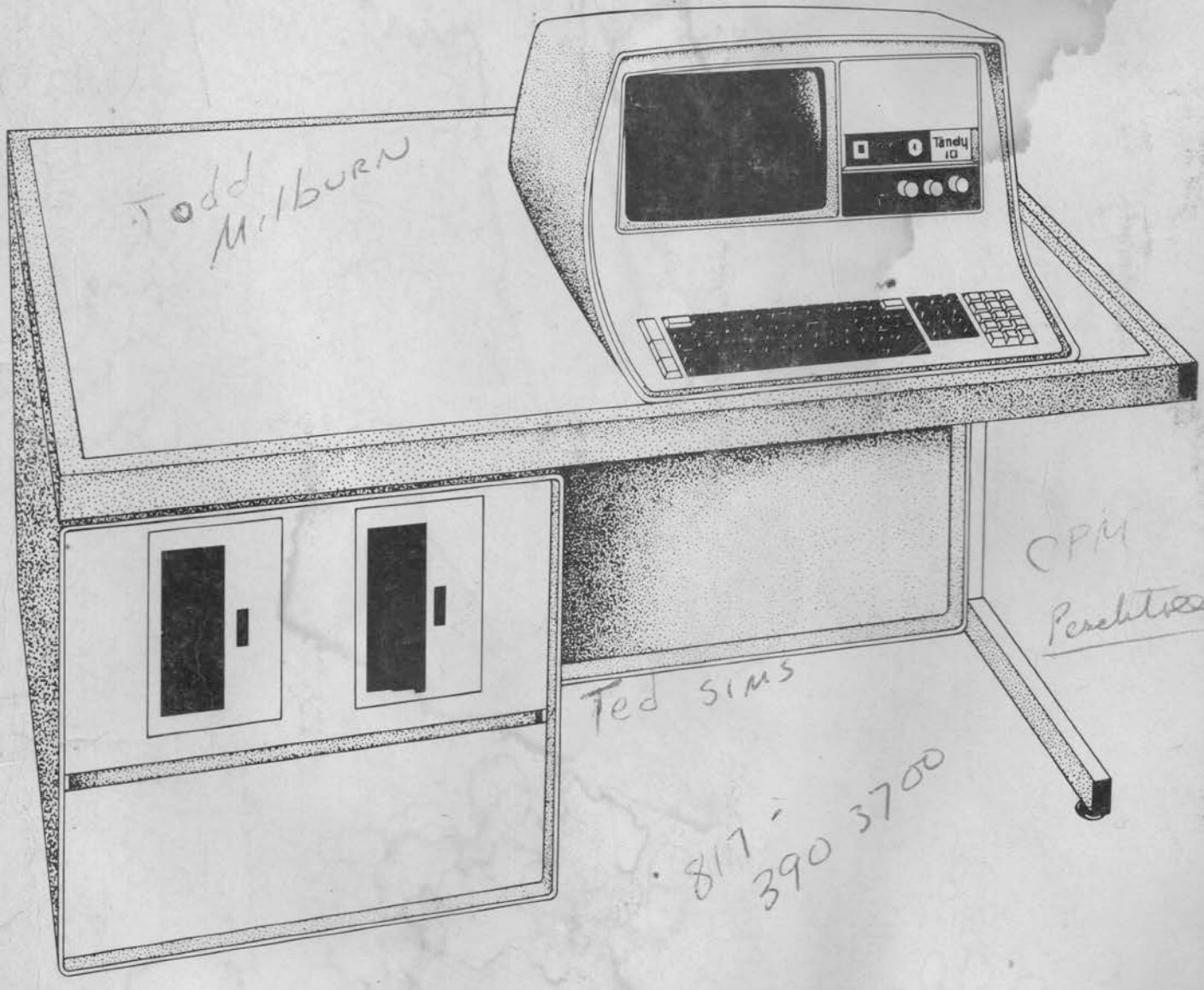
BASIC Reference Manual

Tandy 10



Todd Milburn

CPM Perchtree

Ted Sims

817-390 3700

CUSTOM MANUFACTURED FOR

Radio Shack®
COMPUTER CENTER

A DIVISION OF TANDY CORPORATION, 700 ONE TANDY CENTER, FORT WORTH, TX 76102

ADDS*BASIC™

Reference Manual

512-33300

9/78

This revision of the ADDS*BASIC™ Reference Manual supersedes the last issue dated 4/78.

A list of changed pages appears below with a margin bar on affected text pages to indicate the changes.

Page No.

iii
iv
v
vi
vii
1-1
1-2
2-5
2-6
2-7
2-8
2-10
2-12
2-13
2-14
2-16
2-17
2-18
2-19
3-7
3-8
3-9
3-10
3-12
3-13
3-14
3-18
3-20
3-21
3-23
3-25
3-26
3-28
3-29
3-31 (New)
3-32
3-33 (New)
3-34 (New)
3-35 (New)
4-2
4-4
5-1
5-5
5-8
5-9
5-10
5-11
5-12
5-13
5-14
6-1
6-2
6-4
6-11
7-1
7-2 (New)
7-3
7-5
7-6
7-7
7-8
7-9 (New)
8-5
8-10
9-1
9-11
9-12
9-13
9-14
9-15
9-17
9-18
9-19
9-21
9-23 (New)
9-24 (New)
9-25
9-26
9-27
9-28
9-29
9-30
9-31
9-32
9-33 (New)
9-34
9-35
9-36
9-38
9-41
9-43
9-44
9-45
A-1
A-2
A-3
A-4
A-5
A-6
A-7
A-8
C-2
C-4
C-9
C-10
C-12
C-14
C-15
D-1
D-2
D-4
F-1
G-1
G-2
G-4
G-5
H-1
I-1
I-2
J-1
J-2

The material contained in this document is furnished for customer reference only, and is subject to change. The techniques described are proprietary and should be treated accordingly.

©1978
ADDS
Hauppauge, N.Y.

TABLE OF CONTENTS

SECTION		PAGE
1	INTRODUCTION	1-1
	Product Description	1-1
	Highlights	1-1
	Features	1-1
	File Handling	1-2
	Program Development	1-2
	Special Function Keys	1-2
	Disk File Operations	1-3
	How To Use This Manual	1-3
2	PROGRAMMING CONCEPTS	2-1
	Line Format	2-1
	Line Number	2-1
	Statement	2-2
	Multiple Statements on a Single Line	2-2
	A Single Statement on Multiple Lines	2-2
	ADDS*BASIC™ Character Set	2-3
	Alphabetic Characters	2-3
	Numeric Characters	2-3
	Special Characters	2-3
	Alphanumeric Characters	2-4
	Use of Blanks	2-4
	Constants	2-4
	Numeric Constants	2-4
	Integer Constants	2-4
	Single Precision Constants	2-4
	Double Precision Constants	2-6
	Hexadecimal and Octal Constants	2-7
	String Constants	2-7
	Label Constants	2-7
	Variables	2-7
	Numeric and String Variables	2-8
	Examples of Variable Names	2-8
	Expressions	2-10
	Operators	2-10
	Arithmetic Operators	2-11
	Relational Operators	2-13
	Logical Operators	2-14
	Functional Operators	2-16
	ADDS*BASIC™ Numeric Output	2-17
	Accuracy	2-18
	Converting From Single to Double Precision	2-19

SECTION PAGE

3 ELEMENTARY ADDS*BASIC™ STATEMENTS..... 3-1

Conventions..... 3-1
AUTO..... 3-2
NEW..... 3-4
REM..... 3-5
LET..... 3-6
STOP..... 3-7
END..... 3-8
PRINT/LPRINT..... 3-9
INPUT..... 3-13
READ..... 3-15
DATA..... 3-17
RESTORE..... 3-18
GOTO..... 3-19
IF...THEN..... 3-20
FOR/NEXT..... 3-23
Subroutines..... 3-26
GOSUB/RETURN..... 3-26
ON...GOTO/ON...GOSUB..... 3-28
RENUM..... 3-29
RESET..... 3-31
LIST/LLIST..... 3-32
DELETE..... 3-33
EDIT..... 3-34

4 PROGRAMMING WITH ARRAYS..... 4-1

What Are Arrays?..... 4-1
Referencing Arrays and Array Elements..... 4-1
Subscripts..... 4-2
Array Size..... 4-2
Dimensioning Arrays..... 4-2
DIM Statement..... 4-3
ERASE Statement..... 4-4
Entering Data Into Arrays..... 4-5
Printing Data From Arrays..... 4-6

5 PROGRAMMING WITH STRINGS..... 5-1

What are Strings?..... 5-1
Referencing Strings..... 5-1
Dimensioning String Arrays..... 5-1
How to Use Strings..... 5-2
Placing Values in Strings..... 5-2
Assignment Statement..... 5-2
INPUT and READ Statements..... 5-3
LINE INPUT Statement..... 5-5
Operations with Strings..... 5-6
Using Strings in Relational Operations..... 5-6
String Concatenation..... 5-6
Using String Functions..... 5-7

SECTION PAGE

5 Numeric Valued Functions..... 5-7
ASC..... 5-7
FRE..... 5-7
INSTR..... 5-8
LEN..... 5-8
VAL..... 5-9
String Valued Functions..... 5-9
CHR$..... 5-9
HEX$..... 5-10
INPUT$..... 5-10
OCT$..... 5-11
SPACE$..... 5-11
STR$..... 5-12
STRING$..... 5-12
More String Valued Functions..... 5-13
LEFT$..... 5-13
MID$..... 5-13
RIGHT$..... 5-14
Printing Strings..... 5-14

6 FORMATTED OUTPUT..... 6-1

What is Formatted Output..... 6-1
How to Format Output..... 6-1
PRINT USING/LPRINT USING Statements..... 6-2
String Fields..... 6-2
Numeric Fields..... 6-3
Print Functions..... 6-11
TAB..... 6-11
SPC..... 6-11
POS..... 6-12
LPOS..... 6-12
Examples of Formatted Output..... 6-12

7 ADVANCED STATEMENTS AND OPERATORS..... 7-1

Advanced Operators..... 7-1
Integer Division..... 7-1
Modulus Arithmetic..... 7-1
Advanced Statements..... 7-1
CLEAR..... 7-2
DEF..... 7-3
DEFINT/DEFSNG/DEFDBL/DEFSTR/DEFUSR..... 7-5
IF...THEN...ELSE..... 7-6
INKEY..... 7-8
LOCATE..... 7-10
POKE..... 7-12
PEEK Function..... 7-12
SWAP..... 7-13
SYSTEM..... 7-14
WAIT..... 7-15

SECTION PAGE

8 DEBUGGING AIDS..... 8-1

TRON/TROFF..... 8-2
ON ERROR GOTO..... 8-4
The ERR and ERL Variables..... 8-5
RESUME..... 8-7
Example of Error Trapping Subroutine..... 8-8
ERROR..... 8-9
Direct Mode..... 8-10
CONT..... 8-11

9 DISK FILE OPERATIONS..... 9-1

File Names..... 9-1
General Disk File Statements..... 9-1
FILES..... 9-2
NAME..... 9-3
KILL..... 9-4
Program Files..... 9-5
SAVE..... 9-6
LOAD..... 9-7
RUN..... 9-9
MERGE..... 9-10
Data Files..... 9-11
OPEN..... 9-12
CLOSE..... 9-14
Sequential Access Files..... 9-15
INPUT #..... 9-16
PRINT #/PRINT # USING..... 9-18
LINE INPUT #..... 9-20
EOF Function..... 9-21
LOC Function..... 9-22
INPUT$ Function..... 9-23
Random Access Files..... 9-25
Opening a Random Access File..... 9-25
Closing a Random Access File..... 9-25
Random File I/O..... 9-25
Numeric/String Conversion Functions..... 9-27
MKI$/MKS$/MKD$..... 9-27
CVI/CVS/CVD..... 9-27
FIELD..... 9-29
LSET/RSET..... 9-32
PUT 9-34
GET 9-35
LOC Function..... 9-38
LOF Function..... 9-38
EOF Function..... 9-38

SECTION PAGE

9 Using ADDS*BASIC™ with ADDS*PLUS™ Files..... 9-39
Opening an ADDS*PLUS™ File..... 9-39
Closing an ADDS*PLUS™ File..... 9-39
ADDS*PLUS™ File I/O..... 9-39
ADDS+...GET..... 9-40
ADDS+...NEXT..... 9-41
ADDS+...PUT..... 9-42
ADDS+...RESET..... 9-43
EOF Function..... 9-44

APPENDIX A USING ADDS*BASIC™ ON SYSTEM 50/70..... A-1

Start-up Procedures..... A-1
Entering a Program..... A-2
Listing a Program..... A-3
Editing a Program..... A-4
Inserting Lines..... A-4
Deleting Lines..... A-5
Print Form Switch..... A-5
Modifying Lines..... A-6
The EDIT Statement..... A-6
CTRL-A..... A-6
Storing a Program..... A-7
Loading a Program..... A-7
Executing a Program..... A-8
Hard Copy Output..... A-8

B ASCII CHARACTER CODES..... B-1

C ADDS*BASIC™ STATEMENTS..... C-1

D INTRINSIC FUNCTIONS..... D-1

E MATHEMATICAL FUNCTIONS..... E-1

F RESERVED WORDS..... F-1

G ERROR MESSAGES..... G-1

H SPACE AND SPEED HINTS..... H-1

I CONVERTING BASIC PROGRAMS NOT WRITTEN FOR SYSTEM 50 OR SYSTEM 70..... I-1

J SPECIAL FUNCTION KEYS..... J-1

TABLES

Table Page

2.1 Logical Operations..... 2-14
C.1 Summary of Statements..... C-2
D.1 Summary of Intrinsic Functions..... D-1
G.1 Error Messages..... G-2
G.2 Disk Error Messages..... G-5
H.1 Space Allocation..... H-1

PRODUCT DESCRIPTION

ADDS*BASIC™ is an extended version of the BASIC programming language originated at Dartmouth College.

System 50/70 is a diskette-oriented, microprocessor-based intelligent data processing terminal.

ADDS*BASIC™ was developed to give the System 50/70 user the advantage of a simple programming language. For the System 70 user, it integrates this language with System 70's powerful formatting and data entry capabilities. This combination satisfies a wide variety of business, educational and scientific applications.

HIGHLIGHTS

- Easy to learn and use
- Syntax and error checking
- Instant feedback of results
- Reduction of program development time

FEATURES

- Single precision floating point to 7 decimal digits
- Double precision floating point to 17 decimal digits
- Boolean (logical) operations
- String processing with string arrays containing elements of up to 255 characters
- Full range of transcendental functions
- Automatic numbering and renumbering of statement lines
- Amount of usable memory can be noted at any time
- Extensive error handling
- Program tracing
- Access to assembly level subroutines.
- Dynamic allocation and deallocation of matrices at execution time
- Formatted output including asterisk fill, fixed and floating dollar sign, scientific notation, trailing sign and comma insertion

- User-defined functions
- Direct and indirect modes of execution
- Use of special function keys
- Use of the bottom line of the screen as a single line TTY during editing to inhibit scrolling of the upper 24 lines

FILE HANDLING

- Extensive file manipulation capabilities
- Can set up multi-record files
- Sequential access files
- Random access files
- ADDS*PLUS™ files
- IBM 3740 media compatible

PROGRAM DEVELOPMENT

Program development time is appreciably reduced as a result of the many features available to the ADDS*BASIC™ programmer. ADDS*BASIC™ provides for the operation of the computer in two different modes -- indirect and direct. In the indirect mode, ADDS*BASIC™ programs are originally entered at the keyboard, line by line. Depression of the ENTER key stores each line in memory and entire programs may be stored on diskette. They may be recalled and changed, and executed directly on System 50/70 using the CRT screen for visual tracking and the line printer for hard-copy output. In the direct mode, the statements or commands are executed as they are entered into the computer. Results of arithmetic and logical operations are displayed and stored for later use, but the instructions entered in direct mode are not stored in memory, so they may not be saved on disk. However, depression of certain keys permits recovery of the previously entered statement (see Section 8). The direct mode is useful for debugging and for using ADDS*BASIC™ in a "calculator" mode for quick computations which do not justify the design and coding of complete programs.

SPECIAL FUNCTION KEYS

Special function keys permit characters to be inserted or deleted, lines to be inserted or deleted, and characters, lines or the entire screen to be erased. Any screen may be copied on the line printer through a single depression of the PRINT LOCAL key. The CRT screen is 25 lines by 80 characters. The 25th line may be used as a single line TTY or screen by turning the PRINT FORM switch to the ON position. This eliminates the scrolling up of the previous 24 lines so the display remains unaltered while the program is being edited.

DISK FILE OPERATIONS

ADDS*BASIC™ supports both sequential and random access files. These files may be created and accessed through the use of simple BASIC statements. Data files previously created under ADDS*PLUS™ (System 70 only) may be accessed by the ADDS*BASIC™ program for updating, reformatting, calculation and printing.

HOW TO USE THIS MANUAL

The ADDS*BASIC™ Reference Manual is organized for use as both a teaching and a reference manual. The topics are introduced in logical order to facilitate learning BASIC and contain many useful examples. The Appendices provide step-by-step instructions for using ADDS*BASIC™ on System 50/70, and contain alphabetized lists of ADDS*BASIC™ statements, intrinsic functions, reserved words and error messages for quick and easy look-up.

Users familiar with BASIC will quickly develop the skills necessary to use System 50/70. After reading Appendix A, which describes Start-Up Procedures and Editing and provides step-by-step instructions for using ADDS*BASIC™ on System 50/70, the user will be able to begin programming with ADDS*BASIC™ on System 50/70. The following sections will also be of interest to these users:

Section 6	- Lists the format specifications for the PRINT USING statement.
Section 9	- Describes disk file operations.
Appendix J	- Will aid the user in converting BASIC programs which were not written for System 50/70.

Users familiar with System 70 will find that start-up procedures are similar to those used for ADDS*PLUS™.

Users without a knowledge of BASIC will be able to write simple BASIC programs in a few short hours. Sections 2 and 3 explain programming concepts and introduce the user to elementary BASIC statements. Appendix A, which is to be read next, explains the procedure necessary to begin programming on System 50/70. The user should not proceed until these sections have been mastered.

LINE FORMAT

The line is the fundamental unit of an ADDS*BASIC™ program. Each line begins with a line number which corresponds to the address of the line in memory, followed by one or more ADDS*BASIC™ statements.

Form: nnnnn ADDS*BASIC™ statement [:ADDS*BASIC™ statement...]

Depression of the ENTER key terminates each line and stores it into memory.

Line Number (nnnnn)

The line number must be an integer from Ø to 65529. This number indicates the order in which the statements in the program will be executed. Program execution begins with the lowest numbered statement and proceeds in ascending order. It is good programming practice to use increments of 5 or 10 between successive line numbers to allow for insertion of forgotten or additional lines. Lines do not have to be entered in ascending order as ADDS*BASIC™ automatically places each line entered in line number sequence in the program.

```
 5  A = 10                      <ENTER>
10  PRINT "THE VALUE OF A IS"   <ENTER>
20  PRINT A                     <ENTER>
30  END                         <ENTER>
```

These statements are listed and executed in the order 5,10,20,30 regardless of the order in which they were entered. The use of line numbers also enables the normal order of execution (ascending order) to be changed.

```
 5  A = 10                      <ENTER>
10  PRINT "THE VALUE OF A IS"   <ENTER>
15  PRINT A                     <ENTER>
20  A = A-1                     <ENTER>
25  IF A > 0 THEN GOTO 15       <ENTER>
30  END                         <ENTER>
```

In the above example, statements 5-25 are executed in ascending order, then statement 15 is executed since the value of A is 9.

Line numbers may be generated automatically and entire programs may be renumbered in ADDS*BASIC™ by the use of the AUTO and RENUM statements (see Section 3).

Statements without line numbers are executed when the ENTER key is depressed (direct mode) but are not stored in memory.

```
A = 10    <ENTER>
PRINT A   <ENTER>
```

The first statement, A = 10, stores the value 10 in memory and the next statement displays this value on the screen but both statements are lost after execution.

Statement

The first word of each ADDS*BASIC™ statement identifies the type of statement and informs BASIC of the operation to be performed. The list of arguments which follows the identifying word may contain constants, variables, expressions and functions.

1. Multiple Statements on a Single Line

More than one statement can be written on a single line as long as each statement (except the last) is terminated with a colon (:) and the line is no more than 255 characters long. For example:

```
10 PRINT X
```

is a single statement line, while:

```
20 X = 10: PRINT X, Y:  IF Y = 2 GO TO 30
```

is a multiple statement line containing 3 statements.

2. A Single Statement on Multiple Lines

A single statement can be continued on successive lines of a program. To indicate that a statement is to be continued, the line is terminated with the NEW LINE key instead of the ENTER key. For example:

```
100 If A<B + 24          <NEW LINE>
THEN 5                   <NEW LINE>
ELSE PRINT A             <ENTER>
```

and

```
25 W7 = (W-X4*3)*Z-A1/   <NEW LINE>
   (A-B) - 17            <ENTER>
```

The length of a multiple-line statement is limited to 255 characters.

ADDS*BASIC™ CHARACTER SET

The BASIC character set used to represent data constants, variables and expressions is divided into three categories: alphabetic, numeric, and special characters.

Alphabetic Characters

The alphabetic characters in BASIC are the upper and lower case letters of the English alphabet (A-Z, a-z).

Numeric Characters

The numeric characters in BASIC are the digits Ø through 9.

Special Characters

There are 30 special characters in BASIC:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
@	Commercial "at" sign
#	Number or pound sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
{	Left braces
}	Right braces
,	Comma
.	Period or decimal point
'	Single quotation mark
"	Double quotation mark
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
>	Greater than symbol
<	Less than symbol
\	Backslash or integer division symbol
_	Underline

Alphanumeric Characters

The set of all ADDS*BASIC™ characters, the letters, numbers and special characters are known as the set of alphanumeric characters.

Use of Blanks

Blanks may be used freely throughout a program to improve readability.

CONSTANTS

Three types of constants are permitted in ADDS*BASIC™:

1. numeric constants
2. string constants
3. label constants

Numeric Constants

Numeric constants can be positive or negative. ADDS*BASIC™ allows five types of numeric constants:

1. integer
2. single precision
3. double precision
4. hexadecimal
5. octal

Integer Constants

Integer constants are stored in two bytes and must be in the range -32768 to +32767. Commas and decimal points are not permitted.

Examples of Integer Constants

```
   12
  +285
  -368
 25786
```

Single Precision Constants

Single precision constants are stored in four bytes and contain up to seven significant digits. The seventh is rounded up. Again, commas are not permitted. A single precision constant takes one of the following forms:

1. a constant containing a decimal point and seven or fewer significant digits in the range -32768 to +32767

2. a constant containing seven or fewer digits outside the range -32768 to +32767

3. a constant followed by an exclamation point (!)

4. a constant written in floating point E format

Floating point representation is used for very large and very small numbers, and consists of an integer or fixed point number called a mantissa followed by the letter E and an optionally signed integer. The integer following the E is the exponent or power of 10 by which the mantissa is multiplied. The exponent must be in the range -38 to +38. The floating point E format is

SX.XXXXXXESTT

where

S represents the signs of the mantissa and the exponent (they need not be the same)
X represents the digits of the mantissa
T represents the digits of the exponent

Floating Point Representation				Number
123.456E-6	=	123.456×10^{-6}	=	.000123456
123456E05	=	$123456 \times 10^{+5}$	=	12345600000.
1234.56E+7	=	$1234.56 \times 10^{+7}$	=	12345600000.

The E format representation of numbers is very flexible since a number can be written any number of ways. For example,

1E-3 = .01E-1 = 100E-5 = .001

Examples of Single Precision Constants

```
1.2
+0.0035
-2.0
44368.92
38750
145E-3
2378!
1234.56789!
```

Statements 10-50 below have been entered into memory. The constant in line 10 (32767) is integer; the other constants are single precision.

```
10 N=32767
20 M=32768
30 P=2.00
40 Q=2.1
50 R=1234567
```

All single precision constants containing six or fewer digits and no significant decimal portion are automatically followed by an exclamation point (!). All seven digit constants are automatically rounded to six digits and printed in E format. See the program LISTing below.

```
LIST
10 N=32767
20 M=32768!
30 P=2!
40 Q=2.1
50 R=1.23457E+06
OK
```

Note: The LIST statement (see Section 3) lists the program currently in memory.

Double Precision Constants

Double precision constants are stored in eight bytes and contain up to 17 significant digits. The seventeenth is rounded up. A double precision constant takes one of the following forms:

1. a constant containing more than seven significant digits
2. a constant followed by a pound sign (#)
3. a constant in floating point D format

The floating point D format is

```
SX.XXXXXXXXXXXXXXXXDSTT
```

where

S	represents the signs of the mantissa and the exponent
X	represents the digits of the mantissa
T	represents the digits of the exponent

As with E format, the value of the constant equals the mantissa times ten to the T power.

Examples of Double Precision Constants

```
 1234567890
-123.45678
 123.45678D+06
 1234.567#
```

Statements 10-40 below have been entered into memory. The first constant (25780) is integer; the second (42500), single precision; and the last two are double precision.

```
10 A=25780
20 B=42500
30 C=12345678
40 D=1234567.8D+14
```

All double precision constants except those written in D format are automatically followed by a pound sign (#). See the program listing below.

```
LIST
10 A=25780
20 B=42500!
30 C=12345678#
40 D=1.2345678D+20
OK
```

Hexadecimal And Octal Constants

Hexadecimal (base sixteen) constants are designated by the symbols &H preceding the number. The legal characters are the digits (0-9) and the letters A-F. Octal (base eight) constants are designated either by &O or just the ampersand sign (&) and contain the digits 0-7.

Hexadecimal Constants	Octal Constants
&H7C	&235
&H12F	&O1375

Hexadecimal constants are in the range &HFFFF to -&HFFFF. Octal constants are in the range &Ø to &177777.

String Constants

A string constant consists of a list of valid ADDS*BASIC™ characters enclosed within double quotes. The list may contain from 1 to 255 characters.

Examples of String Constants

"**System 70**"	(spaces count, so this string contains 13 characters)
" "	(a string consisting of 2 blanks)
"$12.95"	(a string containing digits is not the same as a number)

Label Constants

A label constant is an unsigned integer constant between Ø and 65529. Label constants are used in line numbers and appear in branching statements.

VARIABLES

A variable is the name used to reference a value. Unlike a constant, the value associated with a variable may change during program execution. Initially, the value of a variable is zero. The programmer must assign a value to a variable if an initial

value other than zero is required. However, it is good programming practice to assign values to all program variables.

Numeric and String Variables

ADDS*BASIC™ allows integer, single precision, double precision and string variables. All variable names must begin with an alphabetic character (A-Z). This may be followed by either another alphabetic character or a digit (0-9). All variables are assumed to be single precision unless explicityly declared by a type definition statement (see DEF statement, Section 7) or followed by one of the special characters below:

%, !, #, $

These special characters are used to declare the variable type. The type declaration character must be the last character of the variable name.

Type	Symbol
Integer (-32768 to +32767)	% (percent)
Single Precision (up to 7 significant digits)	! (exclamation)
Double Precision (up to 17 significant digits)	# (pound)
String (up to 255 characters)	$ (dollar)

A variable name may contain more than two characters, (three if the last is one of the above symbols); however, all characters after the first two (or three) are ignored. To avoid confusion, it is suggested that a maximum of two characters (three including a special symbol) be used. Also, a variable name may not contain one of the reserved words listed in Appendix F.

Examples of Variable Names

Legal	Illegal
A (single precision)	%C (first character must be alphabetic)
BØ (single precision)	GO (reserved word)
K$ (string)	4P# (first character must be alphabetic)
CH#(double precision)	FNT (first two characters are reserved word)
R4%(integer)	F%O (type declaration character must be last)
F! (single precision)	

The BASIC Interpreter does not distinguish between the following pairs of variable names

AB and ABC	(the "C" is ignored)
A1$ and A12$	(the "2" is ignored)
N1 and N1!	(N1 is assumed to be single precision)

The example below shows that variables AB and ABC are considered identical.

```
LIST
10 AB=3
20 ABC=45
30 PRINT ABC;AB
40 END
OK
RUN
 45  45
OK
```

Note: The RUN statement (see Section 9) executes the program in memory.

The example below shows that the variables A$ and A1$ are distinct but A1$ and A12$ are not.

```
LIST
10 A$="A"
20 A1$="X"
30 A12$="D"
40 A2$="S"
50 PRINT A$,A1$,A12$,A2$
OK
RUN
A             D             D             S
```

ADDS*BASIC™ makes no distinction between upper and lower case variable names. For example

```
LIST
10 xa=10
20 XA=7
30 PRINT xa,XA
OK
RUN
 7             7
OK
```

However, the following variable names are recognized as being different.

N1 and N1# and N1%
CD! and CD# and CD%

An attempt to store a single or double precision constant in an integer variable will result in an OVERFLOW error message at execution. See line 20 of the example below.

```
LIST
10 N%=32767
20 M%=43768!
OK
RUN
OVERFLOW IN 20
OK
```

An attempt to store a double precision constant in a single precision variable will result in truncation of the constant in memory. See the example below.

```
LIST
10 N!=123456789#
20 PRINT N!
OK
RUN
 1.23457E+08
OK
```

EXPRESSIONS

A numeric expression combines constants and variables with operators in such a way that it can be evaluated to produce a single value. Numeric expressions used in BASIC are similar in form to algebraic expressions. Thus, most arithmetic operations can be programmed directly without changing their form.

String expressions, a more advanced topic, are covered in detail in Section 5, which contains a complete discussion of the techniques used to program with strings.

Operators

An operator performs a mathematical or logical operation on one or two values resulting in a single value. In the expression A+B the operator (+) is a binary operation-- one operation between two values. In the expression -D, the operator (-) is called a unary operation--an operation on one value. Four types of operators are used in ADDS*BASIC™: 1) arithmetic, 2) relational, 3) logical, 4) functional.

Arithmetic Operators

Operator	Operation	Expression
+	Addition	A+B
-	Subtraction	A-B
*	Multiplication	A*B
/	Division	A/B
^	Exponentiation	A^B
+	Unary Plus	A+(+B)
-	Unary Minus	A+(-B)

Unlike algebraic notation, implied multiplication does not exist in BASIC. Thus A times B must be written as A*B rather than just AB. The operation of raising a number to a power also requires an explicit operator. Thus A^B is written as A^B.

When more than one operation is to be performed in a single expression, the operations are performed in the following order proceeding from left to right.

Operation	Operator
Unary Plus and Minus	+,- (equal priority)
Exponentiation	^
Multiplication and Division	*,/ (equal priority)
Addition and Subtraction	+,- (equal priority)

The expression A/B+C^D is evaluated as follows:

first, C is raised to the D power

second, A is divided by B

third, A/B, the result of the second operation, is added to C^D, the result of the first operation

The algebraic equivalent of the above BASIC expression is

$$\frac{a}{b} + c^d$$

The division symbol (/) may be considered as extending to the left under all elements until it is terminated by the first + or - sign it reaches or by the end of the expression. Its extension to the right will be terminated by any operation symbol except exponentiation.

Parentheses are used to indicate any other order of evaluation. For example, if the sum of B and C is to be raised to the D power, the expression would look as follows:

A*(B+C)^D

Expressions enclosed within parentheses are evaluated first. When parentheses are nested, the expression in the innermost pair of parentheses is evaluated first.

Algebraic Expressions	BASIC Expressions
$x + \frac{y}{c}$	X + Y/C
$\frac{x + y}{c}$	(X + Y)/C
$\frac{x}{yz}$	X/(Y*Z)
$\frac{xy}{c}$	X * Y/C
$\frac{x - y}{cd^3}$	(X - Y)/(C * D^3)
$\frac{xy}{cd}$	X * Y/(C * D)
$(x^y)^z$	(X^Y)^Z
x^{y^z}	X^(Y^Z)
x^{-y}	X^(-Y) (2 operators may not appear consecutively)
$\frac{a}{x^y} + b$	A/X^Y + B

Integer, single precision and double precision constants and variables may be included in the same expression. Before each operation in the expression is performed, the two operands are examined. If one operand is an integer and the other a single precision value, the integer is converted to single precision and the result is a single precision value. If one operand is a double precision value the other operand is converted to double precision and the result is a double precision value.

```
PRINT 20+123.45
 143.45
OK
PRINT 300+1234567
 1.23487E+06
OK
PRINT 20+9999999
 1E+07
OK
PRINT 2+1.8+123456789
 123456792.8
OK
```

After an expression is evaluated, the results are stored in the format determined by the variable type. In the example below, the integer variable X% contains Ø after the expression 1/2 is evaluated while the single precision variable Y! contains .5

```
LIST
10 X%=1/2
20 Y!=1/2
30 PRINT X%,Y!
OK
RUN
 0              .5
OK
```

Below, the sum of 12 and 3.8 is stored in the integer variable P% as 15 and in the single precision variable P! as 15.8

```
LIST
10 P%=12+3.8
20 P!=12+3.8
30 PRINT P%,P!
OK
RUN
 15             15.8
OK
```

In the example below, the sum of a single precision value and a double precision value is truncated to single precision form when stored in the single precision variable X!

```
LIST
10 X!=123456789#+200000!
20 Y#=123456789#+200000!
30 PRINT X!,Y#
OK
RUN
 1.234657E+08  123656789
OK
```

Relational Operators

Relational operators are most often used to make decisions concerning the flow of control in a program (see IF statement) but are also valid expression operators. They compare the values of two operands to see if they bear the relationship to each other signified by the operator. If the relation is true, the result value is 'true'; if not, the result value is 'false'. In most programs it is convenient to think of relational operators as producing logical values as their result. The value 'true' is assigned the number -1 and 'false' is assigned the number Ø. The table below lists the six binary relational operators.

Operator	Relation Tested	Expression
=	Equality	A=B
<>	Inequality	A<>B
<	Less than	A<B
>	Greater than	A>B
>=	Less than or equal to	A<=B
>=	Greater than or equal to	A>=B

Arithmetic and relational operators may be combined in one expression:

$$(A*B) < (A-C/3)$$

The arithmetic operators are always evaluated first, followed by the relational operators. The above expression is evaluated as

true (-1) if $ab<a-\frac{c}{3}$ and false (Ø) if $ab\geq a-\frac{c}{3}$

Logical Operators

Logical operators are useful in testing multiple relations or performing Boolean operations on 'true' and 'false' (non-zero or Ø) values. Like the relational operators, the result of a logical operator is 'true' or 'false', depending upon the values of its operands. An operand is considered to be 'false' if its value is Ø or 'true' if its value is not Ø. The table below describes the action of the logical operators.

Table 2.1. Logical Operations

Operator	Meaning	Expression
NOT (negation)	'true" if operand is 'false'; 'false' if operand is 'true'	NOT A
AND (disjunction)	'true' if both operands are 'true'; 'false' if either operand is 'false'	A AND B
OR (conjunction)	'true' if either operand is 'true'; 'false' if both operands are 'false'	A OR B
XOR (exclusive)	'true' if both operands are different; 'false' if both operands are alike	A XOR B

Table 2.1 Logical Operations (Continued)

Operator	Meaning	Expression
EQV (equivalence)	'true' if both operands are alike; 'false' if operands are different	A EQV B
IMP (implication)	'true' if both operands are alike or first operand is 'false' and second is 'true' 'false' if first operand is 'true' and second is 'false'	A IMP B

Arithmetic, relational and logical operators may be combined in one expression.

A + B OR C = D

The arithmetic operators are evaluated, followed by the relational operators for values of zero (false) or non-zero (true). The above expression is evaluated as Ø (false) only if A + B = Ø and C is not equal to D. Otherwise it is evaluated as true.

The logical operators convert single precision arguments into integers in the range -32768 to +32767. Arguments outside this range cause an OVERFLOW error. After the operations are performed, the result is returned in the same form and range. Truth tables for these operators appear below.

AND

X	Y	X AND Y
1	1	1
1	Ø	Ø
Ø	1	Ø
Ø	Ø	Ø

XOR

X	Y	X XOR Y
1	1	Ø
1	Ø	1
Ø	1	1
Ø	Ø	Ø

OR

X	Y	X OR Y
1	1	1
1	Ø	1
Ø	1	1
Ø	Ø	Ø

EQV

X	Y	X EQV Y
1	1	1
1	Ø	Ø
Ø	1	Ø
Ø	Ø	1

NOT

X	NOT X
1	Ø
Ø	1

IMP

X	Y	X IMP Y
1	1	1
1	Ø	Ø
Ø	1	1
Ø	Ø	1

Logical operations are typically used for 'masking' or testing a binary number for some predetermined pattern of bits.

These examples will serve to show how the logical operations work:

```
61              111101                      12              1100
18              010010                      14              1110
16 AND 18       010000 = 16                 12 AND 14       1100 = 12

-1              1111 1111 1111 1111          4               100
 9              0000 0000 0000 1001          3               011
-1 AND 9        0000 0000 0000 1001 = 9      4 OR 3          111 = 7

-1              1111 1111 1111 1111         11              1011
-2              1111 1111 1111 1110         11              1011
-1 OR -2        1111 1111 1111 1111 = -1    11 OR 11        1011 = 11
```

NOT 0=-1 The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.

NOT X=-(X+1) The two's complement of any number is the bit complement plus one.

Functional Operators

A function names an operation that, when performed, produces a single value result. A numeric function is identified by a variable name followed by one or more arguments in parentheses. Any number of arguments may be used and each argument is a numeric expression.

Since a function results in a single value, it can be used anywhere in an expression where a constant or variable can be used. To use a function, the function name followed by arguments in parentheses is placed in an expression. The function is evaluated and the result is used in the evaluation of the expression.

Examples of common functions:

SQR(X) where X is a numeric expression that results in a value $\geq$ 0. When called, it returns the square root of X. For instance, if N=2, SQR(N+2) = 2.

ABS(X) where X is any numeric expression. When called, it returns the absolute value of X. For instance, ABS(-33) = 33.

These functions are predefined or intrinsic. They are recognized automatically by the system and are available to any program. A complete list of the intrinsic functions for ADDS*BASIC™ is provided in Appendix D. In addition to the predefined functions, the user may define his own functions with the DEF statement (See Section 3). User-defined function names always begin with the letters FN followed by a legal variable name.

The following statement defines a function (FNC) that finds the area of a circle from its radius.

```
10 DEF FNC(R) = 3.1416*R^2
```

Once defined, the function can be used in an expression like any of the predefined functions.

String functions which return a string value are identified by a variable name followed by a $. The functions used in string manipulation or that return a string value are described in Section 5 and summarized in Appendix E.

ADDS*BASIC™ NUMERIC OUTPUT

The format of a printed number is determined by the following rules:

1. If the number is negative, a minus sign (-) is printed to the left of the number. If the number is positive, a space is printed to the left of the number.

2. If the absolute value of the number is an integer in the range 0 to 999999, it is printed as an integer.

3. If the absolute value of the number is greater than or equal to .01 and less than or equal to 9999999999999999, it is printed in fixed point notation with no exponent.

4. If the number does not fall under categories 1, 2 or 3, scientific notation is used.

The formats of scientific notation are as follows:

```
SX.XXXXXESTT              single precision
SX.XXXXXXXXXXXXXXDSTT     double precision
```

where

S represents the signs of the mantissa and the exponent

X represents the digits of the mantissa and

T represents the digits of the exponent

One non-zero digit is printed before the decimal point. Leading zeros are never printed; i.e. the digit before the decimal is never zero. Also, trailing zeroes are never printed. If there is only one digit to print after all trailing zeroes are suppressed, no decimal point is printed. The exponent sign will be "+" for positive and "-" for negative. Two digits of the exponent are always printed; that is, zeroes are not suppressed in the exponent field. The value of any number expressed thus is the number to the left of the "D" or "E" times 10 raised to the power of the number to the right of the "D" or "E." The exponent must be in the range -38 to +38. The largest number that may be represented in ADDS*BASIC™ is 1.70141E38; the smallest positive number is 2.9387E-38. Examples of numbers as input and as output by ADDS*BASIC™ follow:

Number Input	ADDS*BASIC™ Output Format
+1	1
-1	-1
6523	6523
1E20	1E20
-12.34567E-10	-1.23457E-09
1.234567E-7	1.23457E-07
1000000	1E+06
.1	.1
.01	.01
.000123	1.23E-04
-25.460	-25.46

Accuracy

Numeric calculations, whether made by hand or by computer, may be exact but are more often approximate. Approximations are introduced into calculations when digits are omitted by rounding or truncation.

Additional approximations may be introduced when decimal numbers (numbers containing a significant fractional portion) are converted to their internal representations in binary form, if the binary number repeats. Conversion to and from floating point accounts for a minor inaccuracy which is within .00000436%.

The discarding of excess digits is known as round-off errors. To avoid this problem entirely, programmers may wish to use integers for all calculations and insert a decimal point before printing output (see PRINT USING statement, Section 6). To reduce the probability of round-off inaccuracy, double precision values may also be used.

Converting From Single to Double Precision

Single precision decimal numbers that repeat in binary form have more significant digits than will be printed. For this reason, a double precision variable set to a single precision value may not produce the expected result. In statement 20 below, the single precision constant 5.6 is stored in a double precision variable (X#) as 5.599999904632568.

```
LIST
10 X=5.6
20 X#=5.6
30 PRINT X,X#
OK
RUN
 5.6            5.599999904632568
OK
```

If the constant is declared double precision, as shown below, it will produce accurate results.

```
LIST
10 X#=5.6#
20 PRINT X#
OK
RUN
 5.6
OK
```

Similarly, the CDBL function (see Appendix D) which converts the argument to double precision may also not produce the expected result.

```
LIST
10 X=5.6
20 X#=CDBL(X)
30 PRINT X,X#
OK
RUN
 5.6            5.599999904632568
OK
```

However, single precision numbers can be converted to double precision using the string functions VAL and STR$. With these functions, described in detail in Section 5, output of single and double precision numbers will be identical.

SECTION 3:

ELEMENTARY ADDS*BASIC™ STATEMENTS

This section describes the simplest forms of the more elementary ADDS*BASIC™ statements. Some, like AUTO and NEW, are normally used only in direct mode. Others, such as DATA and REM, are normally used only in indirect mode. Remember, statements entered in direct mode are executed immediately and statements entered in indirect mode (with line numbers) are stored in memory.

These statements are sufficient, by themselves, for the solution of most problems. Once these statements are mastered, the user can investigate the more advanced applications of these statements and the additional statements and features explained in the subsequent sections.

CONVENTIONS

The statements described in this manual are presented as follows:

General Form:	Words printed in capital letters must be written exactly as shown. Items enclosed in square brackets [] are optional. Items in lower case letters must be supplied by the user.
Purpose:	A brief statement which describes the operation performed by the statement.
Remarks:	Pertinent comments related to the use of the statement. Any rules are included under this heading.
Examples:	Typical uses are provided to help describe the BASIC verb and its format.

AUTO

General Form:

```
AUTO
AUTO initial line number [,]
AUTO [initial line number], increment
```

initial line number:
number of the next statement to be entered

increment:
distance between each new line number

Purpose:

To generate line numbers automatically when entering program lines.

Remarks:

Every input line will be numbered by the AUTO statement until CTRL-C is depressed.

If the increment is omitted, it is assumed to be 10. If the initial line is omitted, it is also assumed to be 10 unless the increment is present. The initial line then defaults to Ø. If a comma follows the initial line and no increment is specified, the increment used in the last AUTO statement is assumed. An ILLEGAL FUNCTION CALL error will result if no AUTO statement was previously entered.

If AUTO generates a line number that already exists in the program residing in memory, the number is printed followed by an asterisk. This will warn users that any statement keyed in will replace the existing line.

Example 1:

```
AUTO
10 X=24        'INITIAL LINE #=10
20 Y=5         'INCREMENT = 10
30
.
.
90^C            <Depress CTRL-C>
OK
```

Example 2:

```
AUTO 100,20
100 INPUT X,Y          'INITIAL LINE #=100
120 PRINT X,Y          'INCREMENT = 20
140
.
.
260^C                   <Depress CTRL-C>
OK
```

Example 3:

```
AUTO 50,
50 A=3                 'INITIAL LINE #=50
70 B=4                 'INCREMENT = 20
90 PRINT SQR(A 2+B 2)
110
.
.
190^C                   <Depress CTRL-C>
OK
```

Example 4:

```
AUTO ,15
0 X=10                 'INITIAL LINE #=0
15 Y=26                'INCREMENT = 15
30
.
.
165^C                   <Depress CTRL-C>
OK
```

NEW

General Form: NEW

Purpose: To delete the current program and clear all variables.

Remarks: The NEW statement must be used before entering a NEW program. However, it is not required before the first program entered after the BASIC Interpreter has been invoked.

Note: If a program has been previously entered in memory with line numbers 10, 20, ..., 100 and the AUTO statement is keyed in without first deleting this program, the BASIC Interpreter will place an asterisk after the generated line numbers 10, 20, ... 100. The asterisk indicates that these line numbers are presently in use. Depress CTRL-C to terminate the AUTO statement and recapture the previously entered program or enter ADDS*BASIC™ statements to overwrite the stored program.

REM

General Form: REM [remark]

Purpose: To insert explanatory comments within a program.

Remarks: REM statements are ignored when the program is executing. However, the REM statement is stored with the program and is output exactly as entered when LISTed.

REM statements entered in direct mode have no effect. If control is transferred to a REM statement from a GOTO or GOSUB statement, then execution continues with the next executable statement following the REM statement. If no executable statement follows the REM statements then the program will act as though an END statement were encountered and control will return to the command level of BASIC.

Example:

```
LIST
10 REM -- REMARKS THROUGHOUT A PROGRAM HELP TO
20 REM -- EXPLAIN THE PURPOSE OF STATEMENTS
30 REM -- LINES 10,20 AND 30 ARE NOT EXECUTED
40 REM -- THIS "PRINT" STATEMENT IS NOT EXECUTED:  PRINT "HELLO"
50 PRINT "BYE": REM -- THIS "PRINT" STATEMENT IS EXECUTED
OK
RUN
BYE
OK
```

Note: Remarks may also be added to the end of a program line by separating the statement and remark with a single quotation mark (') in place of the letters REM.

Example:

```
LIST
10 PRINT X     'THIS IS A REMARK
OK
```

LET or Assignment

General Form: [LET] variable = expression

variable:
numeric or string variable name
expression:
numeric or string expression

Purpose: To assign a value to a variable.

Remarks: The BASIC verb LET is optional.

The equal sign does not indicate algebraic equality but is a signal that the value of the expression on the right is to be assigned to the variable on the left.

Examples:

```
10 A=12        '12 IS ASSIGNED TO VARIABLE A
20             '
30 A=A+1       'VARIABLE A IS ASSIGNED A VALUE
40             'ONE GREATER THAN IT WAS BEFORE
50             '
60 D=SQR(20)   'VARIABLE D IS ASSIGNED THE
70             'VALUE OF THE SQUARE ROOT OF 20
```

Note: For examples of assignment statements using subscripted variables and string variables, see Sections 4 and 5 respectively.

STOP

General Form: STOP

Purpose: To terminate execution of the program and return control to the command level of BASIC.

Remarks: STOP statements may be placed anywhere in the program to terminate execution. When STOP is encountered in the program, the system will display the following message on the screen:

```
BREAK IN (line) NNNNN
```

A STOP statement may be inserted in a program for debugging purposes. After program execution has been terminated, direct mode statements can be entered to print intermediate values or change the values of variables. Program execution can be resumed with the CONT statement (see Section 8).

Unlike END, the STOP statement does not close files. (See Section 9 for a discussion of disk file I/O.)

Example:

```
AUTO
10 X=10
20 PRINT X
30 STOP
40 ^C
OK
RUN
 10
BREAK IN 30
OK
```

See IF statement for an example of multiple STOP statements in the same program.

END

General Form: END

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

Remarks: Only one END statement is permitted in each program. If used, the END statement must be the last statement of the program.

The END statement closes all disk files. (See Section 9 for a discussion of disk file I/O.)

Example:

```
AUTO
10 X=10
20 PRINT X
40 END
50 ^C
OK
RUN
 10
OK
```

Note: The system does not display a message when program execution is terminated by an END statement.

PRINT/LPRINT

General Form: PRINT [print list]
LPRINT [print list]

print list:
a combination of one or more numeric expressions and string expressions separated by commas or semicolons

Abbreviation: The question mark (?) may be used as an abbreviation for the BASIC verb PRINT. However, LPRINT may not be abbreviated by entering L?.

Purpose: To output data on the screen/printer.

Remarks: If the optional print list is missing, the effect is to skip 1 line.

Zone Spacing of Output

The print line on the screen is divided into five print zones and the print line on the printer into nine print zones. The width of a print zone is 14 spaces. The first column number on a line is column 1.

```
|1      14|15      28|29      42|43    56|57     70|
|   14    |   14     |   14     |  14    |   14    |
| columns| columns  | columns |columns| columns|
| Zone 1 | Zone 2   | Zone 3  |Zone 4 | Zone 5 |
```

The position of printing is determined by the punctuation used to separate the entries in the print list.

A comma (,) between items in the print list causes the next item to be printed in the leftmost position of the next printing zone. If there are no more printing zones on the current line, printing continues in the first printing zone on the next line. If a list element requires more than one print zone, the next item in the list is printed in the next free print zone (see example 1).

Before a numeric item is printed its length is compared with the space remaining on the line. If insufficient space is left on the current line the item is moved to the next line. If a string item cannot be printed in its entirety, it is continued on the next line.

A semicolon (;) or space causes the next printing to begin immediately after the last value printed. Numbers are separated by one space, strings are not (see examples 2 and 4).

If a comma or semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line according to the condition above. The absence of any punctuation at the end of the print list causes the next PRINT statement to begin printing on the next line.

Printing String Constants

String constants may be printed if they are enclosed in double quotation marks (").

Printing Numbers

Numbers (integer, decimal, and D-type or E-type) are printed in the following form:

sign number space

The sign is either minus (-) or blank for plus and the number is always followed by a space. (See Section 2 for further details on numeric formats).

Example 1:

```
LIST
10 X=25
20 PRINT "THE SQUARE ROOT OF X IS:",SQR(X)
30 PRINT
40 PRINT "12345678901234567890123456789 0"
50 PRINT "       COLUMN POSITIONS"
60 END
OK
RUN
THE SQUARE ROOT OF X IS:     5

12345678901234567890123456789 0
       COLUMN POSITIONS
OK
```

Line 30 outputs a blank line before the print statements used to denote the column positions. Notice that the square root of X is printed in the next free print zone, Zone 3.

Note: The LIST statement (Section 3) lists the program currently in memory and the RUN statement (Section 9) executes the program in memory.

Example 2:

```
LIST
10 X=5
20 PRINT -X;X*2;X*3;
30 PRINT X-25;X-100
40 PRINT
50 PRINT "12345678901234567890"
60 PRINT "  COLUMN POSITIONS"
70 END
OK
RUN
-5  10  15 -20 -95

12345678901234567890
  COLUMN POSITIONS
OK
```

Note that lines 20 and 30 use the semicolon between entries. The trailing semicolon in line 20 causes the values -20 and -95 to be printed immediately following the values in line 20.

Example 3

```
LIST:
10 X=5
20 PRINT X,(X*2)^6,
30 PRINT X^4
40 PRINT
50 PRINT "12345678901234567890123456789O"
60 PRINT "       COLUMN POSITIONS"
70 END
OK
RUN
 5              1E+06          625

12345678901234567890123456789O
       COLUMN POSITIONS
OK
```

Notice that the trailing comma in line 20 causes the value of X^4 in line 30 to be printed in Zone 3 following $(X \times 2)^6$ rather than Zone 1 of the following line.

Example 4:

```
LIST
10 X$="ADDS"
20 Y$="*"
30 Z$="BASIC"
40 PRINT X$;Y$;Z$
50 STOP
OK
RUN
ADDS*BASIC
BREAK IN 50
OK
```

Example 5:

```
LIST
10 Y=7
20 X=5
30 PRINT X;X^2,Y;Y^2
40 PRINT
50 PRINT "12345678901234567890"
60 PRINT "  COLUMN POSITIONS"
70 END
OK
RUN
 5  25          7  49

12345678901234567890
  COLUMN POSITIONS
OK
```

The comma and semicolon spacing characters are both used in line 30.

Example 6:

```
LIST
10 X=5
20 PRINT X,:Y=-23
30 PRINT Y
40 PRINT
50 PRINT "1234567890123456"
60 PRINT "COLUMN POSITIONS"
70 END
OK
RUN
 5            -23

1234567890123456
COLUMN POSITIONS
OK
```

Line 20 above is a multiple statement line. The comma terminating the print list causes the value of Y in line 30 to be printed in Zone 2 rather than Zone 1 of the following line.

Notes:

If LPRINT is substituted for PRINT in the above examples the output would appear on the printer instead of the screen.

The PRINT USING and LPRINT USING statements described in Section 6 give the user greater control over the format of program output.

INPUT

General Form: INPUT ["prompt string";] variable list

variable list:
one or more numeric or string variable names separated by commas
prompt string:
message to be printed at execution time

Purpose: To input data from the terminal during program execution.

Remarks: When an INPUT statement is executed, a question mark (?) is displayed on the screen and the program pauses until the input requirements are satisfied. If the prompt string is included in the INPUT statement, it is printed before the question mark.

The user responds by typing a list of data, where each datum is separated from the next by a comma. The ENTER key is then depressed. If a value has not been supplied for each of the elements of the variable list, a double question mark (??) will be output as a prompt, indicating there are further data list elements which must be supplied. If more data was typed than was requested, the warning ?EXTRA IGNORED is printed and execution proceeds. After all the requested data is input, execution continues at the statement following the INPUT.

The data input in response to the prompt must be of the same mode (numeric or string) as the variable in the INPUT statement list for which the data is being supplied. If data input from the terminal does not match the mode of a variable in the INPUT statement list, then ADDS* BASIC™ prints ?REDO FROM START and waits for the current data to be entered.

Example 1:

```
LIST
10 REM **PROGRAM TO ILLUSTRATE
20 REM   INPUTS DURING EXECUTION**
30 INPUT A,B,C,D,E
40 PRINT A+B,C-D,E
50 STOP
OK
RUN
? 1,2,3,4,5
 3            -1            5
BREAK IN 50
OK
```

Example 2:

```
LIST
10 REM --PROGRAM TO ILLUSTRATE
20 REM   INPUT PROMPT STRING--
30 INPUT "TYPE A NUMBER";A
40 PRINT A,A*A
50 GOTO 30
OK
RUN
TYPE A NUMBER? 3
 3             9
TYPE A NUMBER? -6
-6             36
TYPE A NUMBER? 8
 8             64
TYPE A NUMBER? ^C
BREAK IN 30
OK
```

Example 3:

The statement

```
10 INPUT "YOUR AGE IS";A
```

is equivalent to

```
10 PRINT "YOUR AGE IS";
20 INPUT A
```

Note:

In example 2 above, ^C denotes the depression of CTRL-C to return to the command level of BASIC.

READ

General Form: READ variable list

variable list:
one or more numeric or string variable names separated by commas

Purpose: To read values from the data list (DATA statements) and assign them to the variables listed in the READ statement.

Remarks: READ statements are always used in conjunction with DATA statements and may also be used with RESTORE statements (see following pages).

The variables listed in the READ statement may be subscripted or non-subscripted and may be numeric or string (see Sections 4 and 5).

The order in which variables appear in the READ statement is the order in which values for the variables are retrieved from the data list. A data element pointer is moved to the next available value in the data list as values are retrieved for variables in READ statements. If the number of variables in the READ statement exceeds the number of values in the data list, an OUT OF DATA error message is printed. If there are more values stored in DATA statements than are read by a READ statement, the next READ statement to be executed will begin with the next unread DATA list entry.

A single READ statement may access more than 1 DATA statement and more than 1 READ statement may access the data in a single data statement.

The mode (numeric or string) of the READ statement variable must match the mode of the corresponding DATA element value or a SYNTAX ERROR message is printed.

If a READ statement is entered in the direct mode, an OUT OF DATA error occurs unless a DATA statement was previously entered in indirect mode.

Example:

```
LIST
10 READ A,B,C,D,E$
20 X=A+B
30 Y=C-D
50 DATA 50,75
60 DATA 8.6,9.3,"THE END"
70 PRINT X,Y,E$
80 STOP
OK
RUN
 125          -.7          THE END
BREAK IN 80
OK
```

Note: See DATA and RESTORE statements.

DATA

General Form: DATA constant list

constant list:
a list of numeric or string constants separated by commas (string constants must be enclosed in double quotes)

Purpose: To provide values for variables appearing in READ statements.

Remarks: The DATA statement is a non-executable statement.

The values appearing in a DATA statement or statements form a single list. The first element of this list is the first item in the lowest numbered DATA statement. The last item in this list is the last element in the highest numbered DATA statement.

DATA statements may be placed anywhere in a program except after the END statement. The data items will be read in sequence as required by READ statements.

Example:

```
100 DATA 1,17, "TESTING", 3E-5
```

(See the READ and RESTORE statements for additional examples.)

Note: A DATA statement entered in the direct mode has no effect.

RESTORE

General Form: RESTORE [line number]

line number:
a program statement line number

Purpose: To reset the program's data pointer to the beginning of the data list or to the data statement with the specified line number.

Remarks: After the statement

```
RESTORE
```

is executed, the next item of data accessed by a READ statement will be the first item in the first DATA list in the program. This allows rereading of the data.

Example: The statement in line 70

```
RESTORE 50
```

repositions the data pointer to line 50.

```
LIST
10 REM !PROGRAM TO ILLUSTRATE USE
20 REM  OF RESTORE STATEMENT!
30 READ A,B,C,D
40 DATA 1,2
50 DATA 3,4,5,6
60 PRINT A,B,C,D
70 RESTORE 50
80 GOTO 30
OK
RUN
 1            2            3            4
 3            4            5            6
 3            4            5            6
 3            4            5            6
 3            4            5            6
^C
BREAK IN 60
OK
```

GOTO

General Form: GOTO line number

line number:
a program statement line number

Purpose: To unconditionally transfer control to a statement that is not in normal sequence order.

Remarks: If control is transferred to an executable statement, that statement and those following will be executed.

If control is transferred to a non-executable statement (e.g., DATA), program execution will continue at the first executable statement which follows the non-executable statement.

Example: See INPUT statement for an example of a GOTO statement.

Note: If a GOTO statement is entered in the direct mode, an UNDEFINED LINE NUMBER error will result, unless a statement with the specified line number had previously been entered in the indirect mode.

IF...THEN

General Form:

```
IF expression THEN statement
IF expression THEN line number
IF expression GOTO line number
```

expression:
numeric or string expression as defined in Section 2

line number:
a program statement line number

statement:
any ADDS*BASIC™ statement except FOR, NEXT, DEF, END, DATA, REM, DIM

Purpose:

To transfer conditionally from the normal consecutive order of line numbers, depending on the truth of an expression.

Remarks:

If, after evaluation, the expression is true (non-zero), then the program statement following the THEN is executed or control is transferred to the specified statement number, depending on the general form used.

If the expression is false (zero), program execution continues at the next physical line after the IF...THEN statement.

When multiple statements occur on single line, an IF...THEN statement must be the last statement on that line. If not, when the expression is false, all statements following the THEN are skipped and control transfers to the next physical line instead of the next statement on the current line. See Example 4.

If an IF...THEN statement is followed by a line number in the direct mode, an UNDEFINED LINE NUMBER error will result unless a statement with the specified line number had previously been entered in the indirect mode.

Since the internal representation of decimal numbers may not be exact (see Accuracy, Section 2), it is advisable to test for a range of values when testing for a non-integer. For example, if the result of a computation A was to be 1.0, a reliable test for 1 is

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

If this test is true, then A is equal to 1 within 1 part in 10^6. This is approximately the accuracy of single precision floating point calculations.

Example 1: The following three statements are equivalent. If the expression X=Y+3 is true, BASIC branches to line 70. Otherwise execution proceeds to the next line.

```
10 IF X=Y+3 THEN 70
20 IF X=Y+3 THEN GOTO 70
30 IF X=Y+3 GOTO 70
```

Example 2: In the following lines, the PRINT statements are executed only if the expressions are true (non-zero).

```
10 IF X THEN PRINT X
20 IF A<B+C OR X THEN PRINT X
```

Example 3: The following program will determine whether the number input is even or odd and print out the number with an appropriate message.

```
LIST
10 INPUT "TYPE AN INTEGER NUMBER";N
20 M=N/2
30 K=INT(M)
40 IF M=K THEN 70
50 PRINT N"  IS ODD"
60 STOP
70 PRINT N"  IS EVEN"
80 STOP
OK
RUN
TYPE AN INTEGER NUMBER? 55
 55   IS ODD
BREAK IN 60
OK
```

The INT function returns the largest integer $\leq$ M. For example, INT(5.9)=5 and INT(-2.8) = -3.

No punctuation is used between the "N" and the first quotation mark in statements 50 and 70. On output, one space would separate the numeric value of A and the string literal "IS ODD" or "IS EVEN". The two spaces (ØØ) after the first quotation mark in each statement will provide two additional spaces between the number and the literal.

Example 4:

```
LIST
10 INPUT "ENTER TWO VALUES";X,Y
20 IF X=Y THEN GOTO 50:PRINT "X<>Y"
30 PRINT "ALL DONE"
40 GOTO 10
50 PRINT "X=Y"
60 GOTO 10
OK
RUN
ENTER TWO VALUES? 5,6
ALL DONE
ENTER TWO VALUES? 7,7
X=Y
ENTER TWO VALUES? ^C
BREAK IN 10
OK
```

When X is not equal to Y, control is automatically transferred to line 50. The message "X<>Y" is never printed.

Notes:

See Sections 4 and 5 for examples of IF...THEN statements using arrays and string expressions. An expanded version of the IF...THEN statement, IF...THEN...ELSE, is described in Section 7.

FOR/NEXT

General Form: FOR variable=initial value TO final value [STEP size]

NEXT [variable]

variable:
non-subscripted integer or single precision variable name which must be the same for both statements

initial value:
initial value of the variable

final value:
final value of the variable

step size:
increment added to the variable each time the loop is executed

Purpose: To repeat a group of statements a specified number of times.

Remarks:

Program Loop Operation

The initial value, final value and step size are expressions that are evaluated when the FOR statement is encountered. The variable is set to the initial value and the statements following the FOR statement are then executed. When a NEXT statement is encountered, the increment is added to the variable and the variable is tested against the final value.

If the increment is positive and the variable is less than or equal to the final value, or if the step is negative and the variable is greater than or equal to the final value, then control transfers back to the statement immediately following the FOR statement. Otherwise execution proceeds with the statement following the NEXT. All loops are executed at least once. If the STEP is not specified, it is assumed to be one.

Execution proceeds faster when an integer variable is used.

Nesting Loops

FOR...NEXT loops may be nested. That is, BASIC can execute a FOR...NEXT loop within the context of another loop.

If loops are nested, they must have different loop variable names. The NEXT statement for the inside loop must appear before that for the outside loop.

Legal	Illegal
FOR I=...	FOR I=...
FOR J=...	FOR J=...
NEXT J	NEXT I
NEXT I	NEXT J

Any number of levels of nesting is allowed up to the limit of available memory.

The variable in the NEXT statement is optional. If omitted, it will match the most recent FOR statement.

Example 1:

```
LIST
10 FOR I=1 TO 9.4
20 PRINT I;
30 NEXT I
40 STOP
OK
RUN
 1  2  3  4  5  6  7  8  9
BREAK IN 40
OK
```

This loop executes 9 times and prints the numbers from 1 to 9.

Example 2:

```
LIST
10 FOR B=5 TO 1 STEP -1
20 PRINT B;
30 NEXT
OK
RUN
 5  4  3  2  1
OK
```

This loop has a negative increment.

Example 3:

```
LIST
10 FOR J=1 TO 9 STEP 3
20 PRINT J
30 NEXT
OK
RUN
 1
 4
 7
OK
```

This loop executes 3 times.

Example 4:

```
LIST
10 FOR I=1 TO 3
20 FOR J=1 TO 5
30 PRINT I;J,
40 NEXT J,I
50 PRINT "LOOP COMPLETED", "I"=";I,"J=";J
OK
RUN
 1  1       1  2       1  3       1  4       1  5
 2  1       2  2       2  3       2  4       1  5
 3  1       3  2       3  3       3  4       3  5
LOOP COMPLETED       I=4        J=6
OK
```

This is an example of nested loops. Notice that the inner loop cycles faster than the outer loop.

Notes:

Each variable is incremented before leaving a FOR/NEXT loop. In example 4 above, I=4 and J=6 after the loop has been executed.

A single NEXT statement may be used as the terminal statement of more than one loop (see statement 40 above). However, the first variable in the NEXT statement must be that of the most recent loop; the second, of the next most recent loop, etc.

Example 5:

Round-off errors may occur when a decimal number is used as the increment of a FOR...NEXT loop (see Accuracy, Section 2). The cumulative effect of these round-off errors over a loop executed many times can be significant. In the example below, the decimal number .01 is used as the increment of the loop which was expected to be executed 100 times. However, this loop was actually executed 101 times.

```
LIST
10 Y=0
20 FOR I=0 TO 1 STEP .01
30 Y=Y+1
40 REM  Y COUNTS THE NUMBER OF TIMES
50 REM      THIS LOOP IS EXECUTED
60 NEXT
70 PRINT "THIS LOOP HAS BEEN EXECUTED";Y;"TIMES"
80 PRINT "THE FINAL VALUE OF THE VARIABLE IS";I
OK
RUN
THIS LOOP HAS BEEN EXECUTED 101 TIMES
THE FINAL VALUE OF THE VARIABLE IS 1.01
OK
```

SUBROUTINES

A subroutine is a group of program statements which is entered via the GOSUB statement and exited via the RETURN statement. Rather than repeat the statements at each point they are required, the statements are written into the program only once and are accessed by a GOSUB statement. The RETURN statement allows control to return to the statement following the last GOSUB statement. In this manner, the program continues at the appropriate place after the subroutine has been executed.

GOSUB/RETURN

General Form: GOSUB line number

RETURN

line number:
a program statement line number

Purpose: GOSUB directs program control to the first statement of a subroutine. RETURN exits the subroutine and returns program control to the next statement following the GOSUB statement that caused the subroutine to be entered.

Remarks: A subroutine may be called from more than one place in a program, and a subroutine may contain a call to another subroutine. Such subroutine nesting is limited only by available memory. See Example 2.

A subroutine may have more than one RETURN statement if program logic requires the subroutine to terminate at one of a number of different places.

Although a subroutine may appear anywhere in a program it is good practice to place the subroutine distinctly separate from the main program. In order to prevent inadvertent entry to the subroutine by other than a GOSUB statement, the subroutine should be preceded by a STOP statement or GOTO statement which directs control to a line number following the subroutine.

Example 1:

```
LIST
10 A=6
20 GOSUB 70
30 A=10
40 GOSUB 70
50 STOP
60 REM      ***SUBROUTINE***
70 FOR I=1 TO A STEP 2
80 PRINT I;
90 PRINT
100 RETURN
OK
RUN
 1  3  5
 1  3  5  7  9
BREAK IN 50
OK
```

Example 2:

```
LIST
10 GOSUB 40
20 PRINT "EXAMPLE"
30 STOP
40 PRINT "NEST";    'SUBROUTINE CALLED BY 10
50 GOSUB 80
60 PRINT "INE ";
70 RETURN           'TO LINE #20
80 PRINT "ED ";     'SUBROUTINE CALLED BY 50
90 GOSUB 120
100 PRINT "ROUT":
110 RETURN          'TO LINE #60
120 PRINT "SUB";    'SUBROUTINE CALLED BY 90
130 RETURN          'TO LINE #100
OK
RUN
NESTED SUBROUTINE EXAMPLE
BREAK IN 30
OK
```

ON...GOTO/ON...GOSUB

General Form: ON expression GOTO list of line numbers
ON expression GOSUB list of line numbers

expression:
numeric expression
list of line numbers:
a list of program statement numbers

Purpose: To transfer control to one of several lines in a program depending on the computed value of an expression at the time the statement is executed. The expression is evaluated and if it is not an integer the fractional portion is truncated. The program transfers control to the line number whose position in the argument list corresponds to the computed value of the expression.

If the expression evaluates to an integer that is greater than the number of lines given in the argument list or that is equal to zero, the ON statement is ignored and control passes to the next statement.

If the integer is less than zero, or greater than 255 an ILLEGAL FUNCTION CALL error will result.

The ON...GOSUB statement must contain an argument list whose lines are the first lines of subroutines within the current program.

Example:

```
10 ON M-5 GOTO 500,75,1000
```

If M-5 evaluates to 1, 2 or 3 then control will transfer to statement 500, 75 or 1000, respectively. If M-5 evaluates to any other value greater than or equal to zero, control will transfer to the next sequential BASIC statement in the program.

RENUM

General Form: RENUM [[new line#],[present line #],[increment]]

new line #:
new number of the first line to be resequenced
present line #:
present line number of the first line to be resequenced
increment:
distance between each new line number

Purpose: To renumber an entire program or a segment of a program stored in memory.

Remarks: The RENUM statement allows program lines to be "spread-out" so that a new line or lines may be inserted between existing lines.

The new line #, present line # and increment are all optional. If the new line # is omitted, it is assumed to be 10. If the present line # is omitted, the entire program will be resequenced. If the increment is omitted, it is also assumed to be 10.

Lines less than the present line # will not be renumbered.

All numbers appearing after a GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL are changed by RENUM to refer to the new line numbers. If a line number appears after one of the statements above but does not exist in the program the message "UNDEFINED LINE XXXXX IN YYYYY" is displayed. XXXXX is not changed by RENUM, but line number YYYYY may be changed.

RENUM cannot be used to change the order of program lines nor to create line numbers greater than 65529 or an ILLEGAL FUNCTION CALL error will result. For example, do not enter RENUM 25,40 when the program has 4 lines numbered 10,20,30 and 40.

Example 1:	RENUM	Renumbers the whole program to start at line 10 with an increment of 10 between the new line numbers.
Example 2:	RENUM 100,,20	Renumbers the whole program to start at line 100 with an increment of 20
Example 3:	RENUM 500,400,15	Renumbers the lines from 400 up so they start at 500 with an increment of 15.

RESET

General Form: RESET

Purpose: To construct a new map of the directory.

Remarks: This statement must be entered immediately after switching diskettes in any drive. If omitted, program and/or object files on the newly inserted diskette may be destroyed.

With ADOS Release 1.4 and later, a BDOS error R/O (read only) occurs if an attempt is made to write to a newly inserted diskette without first entering the RESET statement.

The RESET statement, usually entered only in direct mode, performs the same function as the RESET key, but it does not return control to the **Command Processor**. The system remains under the control of the BASIC Interpreter.

LIST-LLIST

General Form: LIST [[starting line #][-[ending line #]]]
LLIST [[starting line #][-[ending line #]]]

Purpose: To list program lines on the terminal (LIST) or on the printer (LLIST)

Remarks: There are several listing options depending on the format of the statement.

1. LIST
   LLIST
   Lists the entire program currently in memory starting with the lowest numbered line.

2. LIST line #
   LLIST line #
   Prints the specified line if it exists.

3. LIST starting line # -
   LLIST starting line # -
   Lists all lines with numbers greater than or equal to the specified starting line number.

4. LIST - ending line #
   LLIST - ending line #
   Lists all lines from the beginning of the program to the specified ending line number inclusive.

5. LIST starting line # - ending line #
   LLIST starting line # - ending line #
   Lists all lines from the starting number to the ending number, inclusive

Listing may be terminated by depressing CTRL-C or halted by depressing CTRL-S. To resume the listing depress any key except CTRL-C.

Example 1:	LIST 200-	Lists all program lines greater than or equal to 200
Example 2:	LIST -400	Lists all program lines from the beginning to line 400
Example 3:	LIST 20-150	Lists all program lines from 20 to 150 inclusive.

DELETE

General Form: DELETE [[starting line #] - [ending line #]]

Purpose: To delete line(s) of the current program -- the program stored in memory.

Remarks: There are several options depending on the format of the statement.

1. DELETE line #

   Deletes only the specified line number. A single line may also be deleted by typing the line number and depressing the ENTER key.

2. DELETE - ending line #

   Deletes all program lines from the beginning of the program to the specified line number.

3. DELETE starting line # - ending line number

   Deletes all program lines from the starting line number to the ending line number, inclusive.

An ILLEGAL FUNCTION call error occurs if the specified line numbers do not exist.

Example 1:	DELETE 350	Deletes line 350.
Example 2:	DELETE - 100	Deletes all program lines from the beginning to line 100.
Example 3:	DELETE 20-140	Deletes all program lines from 20 to 140 inclusive.

EDIT

General Form: EDIT line number

line number:
a program statement line number

Purpose: To modify or correct a statement of the current program -- the program stored in memory.

Remarks: The entire line to be edited will be printed and the cursor will be positioned under the first character of the statement.

The EDIT statement is used in conjunction with the following powerful editing and cursor control function keys:

HOME,->,<-	to position the cursor under the character to be changed.
DEL CHAR	to delete a character at the current cursor position.
INS CHAR	to insert a character at the current cursor position.

The INS CHAR key is an alternate action key, electronically designed so that a single key depression lights the key and sets it until a second depression resets it. To continue editing after inserting characters, depress the INS CHAR key to leave the insert mode. When editing is complete, depress the ENTER key to store the edited line in memory.

Program lines may also be modified or corrected by retyping the entire line.

Example:

```
EDIT 50
50 CORECTING MISTEAKES IS 4ASY AS 1,2,3
```

a. Use the -> key to position the cursor under the R of CORECTING. To enter the insert mode, depress the INS CHAR key. Key in the letter R then depress the INS CHAR key again to leave the insert mode.

b. Position the cursor under the E of MISTEAKES. To delete the letter E, depress the DEL CHAR key.

c. Position the cursor under the 4. To replace the 4 with an E, depress the E key.

d. Depress the ENTER key to store the edited line in memory.

Note: Depression of the special function keys CTRL-A enables the user to recover a previously entered direct mode statement. CTRL-A causes the system to enter the EDIT mode. The INS CHAR key may then be used to add a line number to the statement.

FOR example:

```
FOR N=1 TO 100      <Depress ENTER key>
OK                  <Depress CTRL-A>
FOR N=1 TO 100
```

A. Depress the INS CHAR key and insert a line number
B. Depress the ENTER key to store the statement in memory

When used in conjunction with the EDIT statement, CTRL-A enables the user to move a BASIC statement by changing its line number. For example:

```
EDIT 30             <Depress ENTER key>
30 FOR N=1 TO 100   <Depress ENTER key>
                    <Depress CTRL-A>
FOR N=1 to 100
```

A. Depress the INS CHAR key and insert a line number
B. Depress the ENTER key to store the statement in memory

C. Delete the original line, line 30.

SECTION 4:

PROGRAMMING WITH ARRAYS

WHAT ARE ARRAYS?

At times it is desirable to keep a table of values in memory and use one name to refer to this table.

```
 567.04
 356.91
-212.95
 408.00
-226.18
```

This table is called an array. An array is a set of variables which is known by one name. Each entry in the table is called an element of the array.

REFERENCING ARRAYS AND ARRAY ELEMENTS

Arrays are referenced by an array variable name. The first character of the variable name must be a letter (A-Z).

An array can be one-dimensional or two-dimensional. A one-dimensional array is organized as a column of elements. Each element is referenced by a subscripted variable with a single subscript specifying the element's position within the column.

In the following one-dimensional array, named A, the first element 567.04 is referenced as A(0) and the last as A(4).

```
A(0)     567.04
A(1)     356.91
A(2)    -212.95
A(3)     408.00
A(4)    -226.18
```

A two-dimensional array is organized as a table of elements containing one or more rows and one or more columns of elements. Each element is referenced by a subscripted variable with two subscripts, the value of the first specifying the row and the value of the second specifying the column which contains the element. Within a program all references to elements of a given array must be consistent with that array's dimensionality. Two-dimensional arrays are rectangular; that is, all rows have the same number of columns.

An array M of five rows and three columns can be represented as follows:

Columns

Rows	1	2	3
1	M(0,0)	M(0,1)	M(0,2)
2	M(1,0)	M(1,1)	M(1,2)
3	M(2,0)	M(2,1)	M(2,2)
4	M(3,0)	M(3,1)	M(3,2)
5	M(4,0)	M(4,1)	M(4,2)

Each element of the array is specified by a pair of subscripts separated by commas; the first indicates the row and the second, the column.

In mathematics, a one-dimensional array is referred to as a column vector and a two-dimensional array as a matrix.

An array variable may have more than two dimensions. The number of dimensions is limited by available memory.

SUBSCRIPTS

The smallest subscript is zero. A subscript may also be an expression. When the expression is evaluated it is truncated to integer form. For example, when the pair of subscripts M(I,2*J) where I=4.2 and J=1.4 is evaluated, I is truncated to 4 and 2*J is evaluated and truncated to the integer 2. This pair of subscripts refers to the element (4,2) of array M. Subscripts may be enclosed in parentheses or square brackets.

ARRAY SIZE

Each array has a size, defined as the number of elements it contains. The size of a one-dimensional array is simply the number of elements in its single column. The size of a two-dimensional array is the product of the number of its rows and columns.

DIMENSIONING ARRAYS

Arrays whose number of rows or columns exceeds 11 (maximum subscripts 10) <u>must</u> be dimensioned using a DIM statement. This allows the system to allocate storage space for the array elements. If no DIM statement has been executed before an array variable is found in a program, BASIC assumes the variable to have a maximum subscript of 10 (11 elements) for each dimension in the reference. However, it is advisable to dimension all arrays with DIM statements to save space and to facilitate program modifications by the original programmer or another programmer at a later date.

DIM

General Form: DIM dimension list

dimension list:
one or more array names followed by the proper number of subscripts in parentheses

Purpose: To allocate storage for array variables.

Remarks: A DIM statement must be used for array variables whose maximum subscripts are greater than ten (11 elements) and may be used for array variables whose maximum subscripts are less than ten to save space.

Each DIM statement may apply to more than one array variable.

If a DIM statement is used it must appear before any reference is made to the array. In the absence of a DIM statement, an array is assigned the default dimensions by ADDS*BASIC™ (a maximum of ten for each subscript). A subsequent DIM statement will then cause a REDIMENSIONED ARRAY error.

Example:

```
110 DIM A(3), B(2,3,4), C(5,7)
```

Array A has 4 elements; array B is three-dimensional, having 3 rows, 4 columns and five levels or floors. Array C is two-dimensional, having 6 rows and 8 columns.

ERASE

General Form: ERASE array variable list

array variable list:
list of array variable names separated by commas.

Purpose: To eliminate the arrays specified.

Remarks: The specified arrays may be redimensioned and the space made available for other uses.

Example:

```
10 DIM A(100,100)
.
.
.
200 ERASE A
210 DIM X (20,5),Y(30,6)
.
.
.
```

ENTERING DATA INTO ARRAYS

There are several methods of assigning values to arrays. Individual elements can be assigned using the assignment statement:

```
10  A(5) = 26
15  N = 15
20  B(1,9) = N*4.5
```

In addition, individual elements can appear in INPUT or READ and DATA statements.

```
30  INPUT A(1),A(2),A(3)
40  READ B(2,2)
50  DATA 2.5
```

An entire array can be filled using READ and DATA statements or INPUT statements in a FOR...NEXT loop.

```
LIST
10 REM **FILLING AN ARRAY**
20 DIM A(2,3)
30 FOR I=0 TO 2
40 FOR J=0 TO 3
50 READ A(I,J)
60 PRINT A(I,J),
70 NEXT J
80 PRINT
90 NEXT I
100 DATA 1,2,3,4,5,6,7,8,9,10,11,12
110 END
OK
RUN
 1            2            3            4
 5            6            7            8
 9            10           11           12
OK
```

To input the elements from the terminal, the following statements could be used:

```
LIST
10 REM FILLING AN ARRAY AT EXECUTION TIME
20 DIM A(2,3)
30 FOR I=0 TO 2
40 FOR J=0 TO 3
50 INPUT "ENTER ARRAY ELEMENT";A(I,J)
60 NEXT J,I
70 END
OK
```

PRINTING DATA FROM ARRAYS

Individual elements as well as entire arrays are printed using the PRINT statement. All formatting is done according to the specifications under the PRINT statement.

```
LIST
10 REM---PRINTING ARRAYS---
20 DIM B(9,9)
30 FOR I=0 TO 9
40 FOR J=0 TO I
50 B(I,J)=1
60 PRINT B(I,J);
70 NEXT J
80 PRINT
90 NEXT I
100 END
OK
RUN
 1
 1  1
 1  1  1
 1  1  1  1
 1  1  1  1  1
 1  1  1  1  1  1
 1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1
OK
```

When line 60 is executed, 1 element of array B is printed for I=0, 2 for I=1 and so on.

WHAT ARE STRINGS?

Strings are groups of up to 255 alphanumeric characters. They are used to provide program dialog with the user to facilitate data input or to print comments and headings during report generation. Strings are the principal type of data used in applications such as text editing and many data-based applications. For example, the following PRINT statement containing a string constant could be used to print a heading in a report generation program:

```
10 PRINT  "QUARTERLY REPORT"
```

REFERENCING STRINGS

Strings can be used directly as in the statement

```
10 PRINT  "ENTER DATA"
```

or they can be given a name, and then the name can be referenced whenever the string of characters is required.

String constants or literals must be enclosed in quotation marks, except when input via the terminal, and may <u>not</u> themselves contain quotation marks. For example:

```
" "HI THERE" "
```

is an illegal string literal.

A string variable name must end with a dollar sign ($) or be explicitly declared by the type definition statement DEFSTR (see Section 7).

Examples: A$, BO$, C1$, D8$

DIMENSIONING STRING ARRAYS

String arrays may be dimensioned in the same manner as numeric arrays by use of the DIM statement. Each element of a string array is a string which may be up to 255 characters long. The total number of string characters in use at any point in the execution of a program must not exceed the total allocation of string space or an OUT OF STRING SPACE error will result. 100 bytes are reserved for string space. This allocation may be changed by the CLEAR statement (see Section 7).

Example: 100 DIM A$(72), B$(10,10)

String array A$ contains 73 strings. String array B$ contains 11 rows and 11 columns. Each element of B$ is itself a string. B$(3,4) refers to the string that is in the 3rd row and the 4th column of the string array B$.

HOW TO USE STRINGS

Strings may be used in a variety of ways. The first step is to assign a value to the string. Once this has been done there is a variety of operators and string functions that can be used to test or modify the string. When string modification is complete, the string can be printed using the same statements as are used for numeric data.

PLACING VALUES IN STRINGS

Values are assigned to string variables by setting them equal to string data, other strings, or string valued functions. A complete list of string functions follows. A special statement (LINE INPUT) enables the user to assign values without the use of quotation marks.

Assignment Statement

The simplest way to set a string value is to use the LET or assignment statement. Strings can be set equal to literal strings, other string variables, or string valued functions.

Example:

```
LIST
10 X$="THIS IS A STRING"
20 Y$=STRING$(10,45)
30 Z$=X$
40 PRINT X$;Y$;Z$
50 END
OK
RUN
THIS IS A STRING----------THIS IS A STRING
OK
```

In the above example, X$ is set equal to a string literal; Y$, to a string valued function; and Z$ to a string variable. See STRING$ function.

INPUT and READ Statements

String variables can be set equal to string data using the INPUT or READ statements.

Examples:

```
10  INPUT "ENTER A STRING"; A$
```

(Requests a string value from the keyboard and assigns it to A$)

```
20  INPUT BO$,Cl$
```

(Requests two string values assigning the first to BO$ and the second to Cl$)

Note: A comma is used to delimit the strings entered at the keyboard. Therefore, the strings themselves may not contain commas unless they are enclosed in double quotes. In the following example, the string below containing city and state

HAUPPAUGE, NEW YORK

is interpreted as two strings

HAUPPAUGE and
NEW YORK

while the following string is stored in a single string variable:

"HAUPPAUGE, NEW YORK"

Example:

```
LIST
10 INPUT "ENTER 2 STRINGS";A$,B$
15 PRINT
20 PRINT A$,B$
25 PRINT
30 GOTO 10
OK
RUN
ENTER 2 STRINGS? HAUPPAUGE,NEW YORK

HAUPPAUGE      NEW YORK

ENTER 2 STRINGS? ADDS,"HAUPPAUGE, NEW YORK"

ADDS           HAUPPAUGE, NEW YORK

ENTER 2 STRINGS? ^C
BREAK IN 10
OK
```

The READ statement can be used to set a string equal to string data contained in a DATA statement. Strings used in DATA statements are limited to a maximum of 255 characters.

Example:

```
LIST
10 READ B$,N,C$
20 A$=LEFT$(B$,4)
30 Q$=CHR$(N)
40 PRINT A$;Q$;C$
50 DATA "ADDS SYSTEM 70",42,"BASIC"
60 END
OK
RUN
ADDS*BASIC
OK
```

LINE INPUT

General Form: LINE INPUT ["prompt string";]string variable name

prompt string:
message to be printed at execution time

Purpose: To input a whole line to a string variable without use of quotation marks and other delimiters.

Remarks: The prompt string is a string literal that is printed on the terminal before input is accepted. A question mark is not printed unless it is contained in the prompt string. All input from the end of the prompt string is assigned to the string variable. LINE INPUT destroys the input buffer, so the command may not be edited by CTRL-A for re-execution. (See Section 3 and Appendix A).

Example 1:

```
LIST
10 LINE INPUT "ENTER TODAY'S DATE:";A$
20 END
OK
RUN
ENTER TODAY'S DATE:4/24/78
OK
```

Notice that a question mark is not printed after the prompt string.

Example 2:

```
LIST
10 LINE INPUT "ENTER CITY AND STATE: ";A$
20 PRINT A$
30 END
OK
RUN
ENTER CITY AND STATE: HAUPPAUGE, NEW YORK
HAUPPAUGE, NEW YORK
OK
```

Notice that the string HAUPPAUGE, NEW YORK is not enclosed in double quotes although it contains a comma. The LINE INPUT statement assigns the entire string to the variable A$.

OPERATIONS WITH STRINGS

Using Strings in Relational Operations

Strings and string variables can be used with relational operators in the same manner as numeric values. The relational operators (=,<,>,<=,>=,<>) can be used with strings to perform branching operations. These operators are also useful for sorting strings.

Strings are compared character-by-character, on the basis of ASCII codes until a difference is found. If the end of one string is reached, while comparison is proceeding, the shorter string is considered to be smaller. ASCII codes may be found in Appendix B.

Examples:

```
A<Z        ASCII A is 065, Z is 090
1<B        ASCII 1 is 049, B is 065
"ØB">"B"   Leading and trailing blanks (Ø)
             are significant in string literals.
```

In the example below, string 123 is less than string AB.

Example:

```
LIST
10 B$="AB"
20 INPUT A$
30 IF A$>B$ THEN 70
40 INPUT C$
50 IF C$<>"ABC" THEN 70
60 PRINT A$ " < AB AND C$ = ABC"
70 END
OK
RUN
? 123
? ABC
123 < AB AND C$ = ABC
OK
```

String Concatenation

Strings may be concatenated (put or joined together) through the use of the "+" operator.

Example:

```
LIST
10 A$="ADDS"
20 B$="SYSTEM 50"
30 C$="70"
40 X$=A$+" "+B$+"/"+C$
50 PRINT X$
60 END
OK
RUN
ADDS SYSTEM 50/70
OK
```

Using String Functions

There are several intrinsic functions that perform special string operations. Some return string values, others return numeric values.

Numeric Valued Functions

The following 5 functions return a numeric value:

Function	General Form
ASC	ASC(X$)
FRE	FRE(X$)
INSTR	INSTR(I,X$,Y$)
LEN	LEN(X$)
VAL	VAL(X$)

where X$,Y$ are string variables, expressions or literals and I is an integer in the range 1 to 255.

ASC Function Form: ASC(X$)

The ASC function converts the first character of the string to a numeric value that is its ASCII code equivalent. A complete list of ASCII characters and their numeric code equivalents is given in Appendix B.

Example:

```
LIST
10 A$="STRING"
20 N=ASC(A$)      'FIRST CHARACTER OF
30 PRINT N        'STRING IS S
40 END            '83=ASCII CODE FOR S
OK
RUN
 83
OK
```

FRE Function Form: FRE(X$)

The FRE function followed by any legal string variable name returns the number of bytes of remaining free string space.

Example:

```
CLEAR 300
OK
PRINT FRE(S$)
 300
OK
```

INSTR Function Form: INSTR([I,]X$,Y$)

The INSTR function searches a string for an occurrence of an imbedded string. This is a powerful tool in any text editing or other word processing application. The INSTR function uses two string arguments, a string to be searched, X$, and a string to look for, Y$. If the "looked for" string is found, the function is set to the position in the "searched" string where the first matched character occurs. If the "looked for" string does not occur, the function is set to Ø. If I>LEN(X$), or if X$ is null or if Y$ can not be found, INSTR returns Ø. If Y$ is null, INSTR returns 1. The optional offset I sets the position for starting the search. I must be in the range of 1 to 255. If the offset I is omitted, the form of this function is INSTR(X$,Y$).

In the example below, S$ is a long string that may contain one or more occurrences of another string R$.

Example:
```
LIST
10 S$="THIS IS THE POINT TO INPUT THE DATA"
20 R$="THE"
30 REM - I IS THE POSITION IN S$ WHERE R$ BEGINS
40 I=1
50 P=INSTR(I,S$,R$)    'P= POSITION OF LOOKED FOR STRING
60 IF P=0 THEN 100     'IF P=0 STRING NOT FOUND
70 I=1+P               'RESET I, CONTINUE SEARCHING
80 PRINT "SEARCH STRING FOUND AT";P
90 GOTO 50
100 END
OK
RUN
SEARCH STRING FOUND AT 9
SEARCH STRING FOUND AT 28
OK
```

LEN Function Form: LEN(X$)

The LEN function returns a numeric value that is the length (number of characters) currently assigned to a string variable. In the example that follows, N is set to the number of characters assigned to (not dimensioned for) A$.

Example:
```
LIST
10 A$="ABC"
20 N=LEN(A$)
30 PRINT "N="N
40 END
OK
RUN
N= 3
OK
```

This would result in N being set to 3. The LEN function is useful in sorting or searching strings or in appending or modifying strings.

VAL Function Form: VAL(X$)

The VAL function returns the numeric value of a string or partial string. The VAL function stops when the first non-numeric character is reached. If the first character is not +,- or a digit, the VAL of the string is zero.

Example:

```
LIST
10 N$="12.96ABC"
20 X=VAL(N$)          'X=12.96
30 READ Y
40 PRINT X+Y          '12.96 + 1.04
50 STOP
60 DATA 1.04
70 END
OK
RUN
 14
BREAK IN 50
OK
```

The VAL function is useful in converting numbers stored as ASCII characters to numeric form so they can be used in arithmetic operations.

String Valued Functions

The following 7 functions return a string value:

Function	General Form
CHR$	CHR$(I)
HEX$	HEX$(X)
INPUT$	INPUT$(X)
OCT$	OCT$(X)
SPACE$	SPACE$(I)
STR$	STR$(X)
STRING$	STRING$(I,J)
	STRING$(I,X$)

where X is a numeric expression, X$ is a string constant or variable and I and J are integer expressions.

CHR$ Function Form: CHR$(I)

CHR$(I) returns a string whose one character is that with ASCII code I.

The statement below will cause a bell to ring on the terminal, since the ASCII code for BEL is 7.

```
PRINT CHR$(7)
```

The statement below will cause only the string WELCOME TO ADDS to be printed in reverse video blink since the ASCII code for reverse video blink is 176 and the ASCII code for full intensity is 164. See Appendix B for the ASCII codes for ADDS visual effects.

```
PRINT CHR$(176); "WELCOME TO ADDS"; CHR$(164)
```

HEX$ Function Form: HEX$(X)

The HEX$ function returns a string which represents the hexadecimal value of the decimal argument X when X is in the range -32767 to +65535

Example:

```
LIST
10 PRINT HEX$(32)
20 END
OK
RUN
20
OK
```

INPUT$ Function Form: INPUT$(X)

The INPUT$ function returns a string of X characters input from the screen during program execution. This function is similar to the INPUT statement, however no prompt is printed on the screen and the characters entered are not echoed back to the screen. All control characters are passed through except CTRL-C, which is used to interrupt execution of the INPUT$ function. If X is not an integer, it is truncated to integer form.

```
LIST
10 PRINT "ENTER A 4 DIGIT NUMBER"
20 A$=INPUT$(4)
30 N=VAL(A$)*.1
40 PRINT "YOUR NUMBER/10 =";N
50 END
OK
RUN
ENTER A 4 DIGIT NUMBER
YOUR NUMBER/10 = 123.4
OK
RUN
ENTER A 4 DIGIT NUMBER
YOUR NUMBER/10 = 780.1
OK
```

```
LIST
10 FOR I=1 TO 5
20 PRINT "ENTER";I;"CHARACTERS"
30 A$=INPUT$(I)
40 PRINT "OUTPUT",,A$
50 NEXT
60 END
OK
RUN
ENTER 1 CHARACTERS
OUTPUT                          1
ENTER 2 CHARACTERS
OUTPUT                          12
ENTER 3 CHARACTERS
OUTPUT                          123
ENTER 4 CHARACTERS
OUTPUT                          1234
ENTER 5 CHARACTERS
OUTPUT                          12345
OK
```

This function may also be used to read X characters from a sequential access file (see Section 9).

OCT$ Function Form: OCT$(X)

The OCT$ function returns a string which represents the octal value of the decimal argument.

Example:

```
LIST
10 PRINT OCT$(32)   'DECIMAL 32=OCTAL 40
20 END
OK
RUN
40
OK
```

SPACE$ Function Form: SPACE$(I)

This function returns a string of spaces of length I and may be used to declare the length of a string or an element of a string array.

Example:

```
LIST
10 A$=SPACE$(4)
20 PRINT "AMOUNT";A$;"PRICE"
30 PRINT "12345678901234"
40 END
OK
RUN
AMOUNT    PRICE
12345678901234
OK
```

STR$ Function Form: STR$(X)

This function returns the string representation of the value of X.

Example:

```
LIST
10 X=56.78
20 Y$="$"+STR$(X)
30 PRINT Y$
OK
RUN
$ 56.78
OK
```

The VAL and STR$ functions should be used to convert single precision numbers to double precision. Use of these functions will assure that double precision numbers will print the same as single precision.

Example:

```
LIST
10 X=2.03
20 X#=X
30 Y#=VAL(STR$(X))
40 Z#=Y#*10
50 PRINT X;X#
60 PRINT Y#;Z#
70 END
OK
RUN
 2.03  2.029999971389771
 2.03  20.3
OK
```

STRING$ Function Form: STRING$(I,J)
STRING$(I,X$)

The first form of this function returns a string of length I whose characters all have ASCII code J. See Appendix B for ASCII Codes.

Example:

```
LIST
10 A$=STRING$(20,42)     'THE ASCII CODE FOR
20 PRINT A$              'THE ASTERISK (*) IS 42
30 END
OK
RUN
********************
OK
```

The second form of this function returns a string of length I containing the first character of X$.

Example:

```
LIST
10 A$=STRING$(20,"ADDS")
20 PRINT A$
30 END
OK
RUN
AAAAAAAAAAAAAAAAAAAA
OK
```

More String Valued Functions

The last three functions are similar in that each returns a partial string of characters from X$. The length of the string is determined by the user.

Function	General Form
LEFT$	LEFT$(X$,I)
MID$	MID$(X$,I,J)
RIGHT$	RIGHT$(X$,I)

LEFT$ Function Form: LEFT$(X$,I)

The LEFT$ function returns the leftmost I characters of string X$.

Example:

```
LIST
10 X$="GOOD MORNING"
20 Y$="NIGHT"
30 Z$=LEFT$(X$,5)
40 PRINT Z$;Y$
50 END
OK
RUN
GOOD NIGHT
OK
```

MID$ Function Form: MID$(X$,I[,J])

The MID$ function returns a string of length J of characters from X$ beginning with the Ith character.

If J is greater than the number of characters in X$ to the right of I, MID$ returns the rest of the string. J is optional. If J is missing, MID$ returns the rightmost characters from X$ beginning with the Ith character.

If I>LEN(X$), MID$ returns the null string.

Example:

```
LIST
10 X$=ABCDEHELLOBYE"
20 Y$=MID$(X$,6,5)
30 Z$=MID$(X$,11)
40 PRINT Y$,Z$
50 END
OK
RUN
HELLO          BYE
OK
```

RIGHT$ Function Form: RIGHT$(X$,I)

The RIGHT$ function returns the rightmost I characters of string X$.

Example:

```
LIST
10 X$=THE SELLING PRICE IS $10.00"
20 Y$=RIGHT$(X$,5)    'Y$=$10.00"
30 C=VAL(Y$)          'C=10.00
40 P=C+C*.07          'P=10.70
50 PRINT P
60 END
OK
RUN
 10.7
OK
```

PRINTING STRINGS

Output of string data is accomplished in the same manner as numeric data. String and numeric data can be mixed in the same output statement. Printing is performed using the PRINT statement.

The PRINT statement allows string data to be output to the terminal.

Example:

```
LIST
10 READ A$,B$,C$
20 PRINT "THIS IS A STRING ";
30 PRINT A$;B$;C$
40 DATA "ASTRING","BSTRING","CSTRING"
50 END
OK
RUN
THIS IS A STRING ASTRINGBSTRINGCSTRING
OK
```

String data can also be output according to a predetermined format with the PRINT USING and LPRINT USING statements (see Section 6).

WHAT IS FORMATTED OUTPUT?

Formatted output gives the user exact control over the format of the program output.

- Numbers can be printed in three different representations: integer, fixed point, and floating point (with scientific notation).
- The exact position of plus and minus signs can be specified permitting construction of neatly aligned numeric tables.
- String values can be printed in specific fields simplifying preparation of reports, graphs and tables.

HOW TO FORMAT OUTPUT

Two statements, PRINT USING and LPRINT USING, enable the user to specify the exact format of the line to be printed on the screen or the printer. In addition, the PRINT and LPRINT statements may be used in combination with two functions, TAB and SPC, to aid in the alignment of output in the form of reports, graphs and tables. Two additional functions, POS and LPOS, return the column position of the screen and line printer. The LOCATE statement, described in Section 7, permits the user to position the cursor at the specified row and column. Together, these statements and functions assist the user in the development of powerful formatting techniques.

See Section 9 for formatting of ASCII file output.

PRINT USING/LPRINT USING

General Form: PRINT USING "format string"; print list
LPRINT USING "format string"; print list

format string:
string variable, expression or constant which specifies the format for printing the items in the list
print list:
list of expressions to be printed; may be separated by commas or semicolons

Purpose: To output on the screen/printer the values of the expressions in the print list using the format specified.

Remarks and Examples: There must be a space between the words PRINT/LPRINT and USING.

All of the characters in the format string will be printed just as they appear, with the exception of the formatting characters. The string will be repeatedly scanned until: 1) the string ends and there are no values in the print list or, 2) a field is scanned in the string, but the print list is exhausted. The string is constructed according to the following rules:

String Fields

! ! specifies a single character string field. (The string itself is specified in the PRINT list.)

\n spaces\ \n spaces\ specifies a string field consisting of 2+n characters. Backslashes with no spaces between them would indicate a field of two characters wide, one space between them would indicate a field three characters wide, etc.

In both cases above, if the string has more characters than the field width, the extra characters will be ignored. If the string has fewer characters than the field width, extra spaces will be printed to fill out the entire field. Trying to print a number in a string field will cause a TYPE MISMATCH error to occur.

Example:

```
LIST
10 A$="ABCDE":B$="FGH"
20 PRINT USING "!";A$,B$
30 PRINT USING "\  \";B$,A$
40 END
OK
RUN
AF
FGH ABCD
OK
```

The formatting character "!" in line 20 specifies a single character string field. Notice that only the first character of each string has been printed. The formatting characters in line 30, backslashes enclosing two spaces, specify four character strings. Notice that a space has been printed for B$ as it has only three characters, while the extra characters of A$ are ignored.

Alphanumeric characters included in the format string will also be printed.

Example:

```
LIST
10 A$="HELLO"
20 PRINT USING "!STRING";A$
30 END
OK
RUN
HSTRING
OK
```

Numeric Fields

With the PRINT USING statement, numeric printouts may be altered to suit almost any application. Strings for formatting numeric fields are constructed from the following characters:

#

Numeric fields are specified by the pound sign (#), each of which will represent a digit position. These digit positions are always filled. The numeric field is right-justified; that is, if the number printed is too small to fill all of the digit positions specified, leading spaces will be printed as necessary to fill the entire field.

The decimal point (.) may be specified in any position in the field. Rounding is performed if necessary. If the field format specifies that a digit is to precede the decimal point (#.###), and the numeric value is less than 1, a Ø will be printed in this position.

Example 1 :

```
LIST
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
OK
RUN
? ##,12
12
? ###,12
 12
? #####,12
   12
? ##.##,12
12.00
? #.###,.12
0.120
? ##.#,3.47
  3.5
? ###,-12
-12
? #.##,-.12
-.12
? ####.#,123
 123.0
? #,6.9
7
? #.#,6.99
7.0
? ^C
BREAK IN 10
OK
```

Note: ^C denotes the depression of CTRL-C to return to the command level of BASIC.

Example 2:

The format string below is a string expression consisting of two string variables and the string concatenation operator "+".

```
LIST
10 A$="###"
20 B$=".##"
30 N=32.45
40 PRINT USING A$+B$;N
OK
RUN
 32.45
OK
```

+ The + sign may be used at either the beginning or end of the numeric field. If the number is positive, the + sign will be printed at the specified end of the number. If the number is negative, a - sign will be printed at the specified end of the number.

Example:

```
LIST
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
OK
RUN
? +#,9
+9
? +#,10
%+10
? +#,-2
-2
? +.###,.12
+.120
? ##+,2
 2+
? ##+,-2
 2-
? ^C
BREAK IN 10
OK
```

If the number to be printed out is larger than the specified numeric field, a % character will be printed followed by the number itself in standard ADDS*BASIC™ format. (The user will see the entire number.) If rounding a number causes it to exceed the specified field, the % character will be printed followed by the rounded number. If, for example, A=.999, then

```
PRINT USING ".##";A
```

will print

```
%1.ØØ
```

If the number of digits specified exceeds 24, an ILLEGAL FUNCTION CALL error results.

- The - sign, when used to the right of the numeric field, will force the minus sign to be printed to the right of the number if it is negative. If the number is positive, a space is printed. A - sign, appearing to the left of the numeric field is treated as a printing character and will always be printed regardless of the sign of the number.

Example:

```
LIST
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
OK
RUN
? ##-,2
 2
? ##-,-2
 2-
? -##,2
- 2
? -##,-2
--2
? -##,-12
-%-12
? ^C
BREAK IN 10
OK
```

**

The ** placed at the beginning of a numeric field causes any leading spaces to be filled with asterisks. The ** also specifies positions for 2 more digits (termed "asterisk fill").

Example:

```
LIST
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
OK
RUN
? **##,1
***1
? **##,12
**12
? **##,1234
1234
? **##,123
*123
? **##,12345
%12345
? **,1
*1
? **,12
12
? **.##,12
12.00
? **#####,1
******1
? ^C
BREAK IN 10
OK
```

$ — A single dollar sign ($) placed at the beginning of a numeric field causes a $ to be printed in the first position of the field -- this is a fixed $.

$$ — When the $$ is used at the beginning of a numeric field, a $ sign will be printed in the space immediately preceding the number printed. This is a floating $. Note that $$ also specifies positions for two more digits, but that the $ itself takes up one of these spaces.

Example:

```
LIST
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
OK
RUN
? $####.##,12.34
$  12.34                    (note: fixed $)
? $$####.##,12.34
   $12.34                   (note: floating $)
? $##,123
$%123
? $$.##,1.23
$1.23
? $$####.##-,-12.34
   $12.34-
? $$####.##,-12.34
  -$12.34
? $####.##,-12.34
$ -12.34
? ^C
BREAK IN 10
OK
```

**$ — The **$ at the beginning of a numeric field causes a dollar sign to be printed immediately preceding the number (floating $) and leading spaces to be filled with asterisks (asterisk fill). The **$ allows for three additional digit positions, one of which is the $ sign.

Example:

```
LIST
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
OK
RUN
? **$####.##,1.23
*****$1.23
? **$.##,1.23
*$1.23
? **$,123
%$123
? **$##,1234
$1234
? ^C
BREAK IN 10
OK
```

,

If commas are required in the numeric field, the format string must be enclosed in double quotation marks.

A comma appearing to the left of the decimal point in a numeric field designation will cause a comma to be printed to the left of every third digit to the left of the decimal point in the number being printed. The comma also specifies another digit position. A comma to the right of the decimal point in a numeric field designation is considered a part of the string itself and is treated as a printing character.

Example 1:

```
LIST
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
OK
RUN
? "#,###.##",1234.56
1,234.56
? "####,.##",1234.56
1,234.56
? "##,##.##",1234.56
1,234.56
? ^C
BREAK IN 10
OK
```

Example 2:

```
LIST
10 INPUT A$,A,B
20 PRINT USING A$;A,B
30 GOTO 10
OK
RUN
? "##.#,"12,13
12.0,13.0
? "##.#, ",12,13
12.0, 13.0
? ^C
BREAK IN 10
OK
```

^^^^

If exponential format is desired in the print-out, the numeric field designation must be followed by ^^^^ to allow space for E+XX or D+XX. Any decimal point arrangement is allowed. The significant digits are left-justified and the exponent is adjusted. Unless a trailing + or - is used, one position to the left of the decimal point is used to print a space or minus sign. Exponential format cannot be used with the floating $ sign if the number to be printed is negative.

Example 1:

```
LIST
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
OK
RUN
? #.###^^^^,123
0.123E+03
? ###.##^^^^,1.23456
 12.35E-01
? ##.#^^^^,12
 1.2E+01
? ##.#^^^^,-12
-1.2E+01
? ##.#^^^^-,-12
12.0E+00-
? ##.#^^^^+,12
12.0E+00+
? $##^^^^,-12
$-1E+01
? $$##^^^^,-12
%-$120E-01
? $$##^^^^-,-12
%$1200E-02-
?^C
```

Example 2:

```
PRINT USING "##^^^^";13,17,-8
 1E+01 2E+01-8E+00
OK
PRINT USING ".######^^^^-";12345,-123456
.123450E+05 .123456E+06-
OK
PRINT USING "+.## ^^^^ ";123,-126
+.12E+03 -.13E+03
OK
PRINT USING "## ^^^^ ####";1000,1234
 1E+03 1234
OK
```

Any alphanumeric characters included in the format string are also printed.

Example:

```
LIST
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
OK
RUN
? THIS IS A NUMBER ##,2
THIS IS A NUMBER  2
? THIS NUMBER IS ODD ###,35
THIS NUMBER IS ODD  35
? BEFORE ## AFTER,12
BEFORE 12 AFTER
? ^C
BREAK IN 10
OK
```

Depressing CTRL-C will terminate the program.

PRINT FUNCTIONS

Two functions, TAB and SPC, are provided to aid the programmer in formatting output with the PRINT and LPRINT statements. These functions may not be used with the PRINT or LPRINT USING statements.

Two additional functions, POS and LPOS, return the current column position of the print head on the screen and the printer.

Function	Format
TAB	TAB(I)
SPC	SPC(I)
POS	POS(I)
LPOS	LPOS(I)

where I is an integer expression in the range 0 to 255.

TAB Function Form: TAB(I)

The TAB function spaces to position I on the screen or the printer. Space 1 is the leftmost space and, for the screen, space 80 is the rightmost. If the column position is already beyond space I, the TAB function has no effect.

Example:

```
LIST
10 PRINT TAB(10);"DATE"
20 PRINT "123456789012"
30 END
OK
RUN
          DATE
123456789012
OK
```

SPC Function Form: SPC(I)

The SPC function is used for spacing. It will print I blanks on the screen or the printer.

Example:

```
LIST
10 PRINT "PRINCIPAL";SPC(8);"INTEREST"
20 PRINT "12345678901234567890"
30 END
OK
RUN
PRINCIPAL        INTEREST
12345678901234567890
OK
```

POS Function Form: POS(I)

The POS function returns the current position of the print head on the screen. The I is a dummy argument and is ignored.

Example:

```
LIST
10 PRINT "HELLO";
20 PRINT POS(0)
30 PRINT "123456789";
40 PRINT POS(0)
50 END
OK
RUN
HELLO 6
123456789 10
OK
```

LPOS Function Form: LPOS(I)

The LPOS function returns the current position of the print head on the printer. The I is a dummy argument and is ignored.

Example:

```
LIST
10 LPRINT "FOUR";
20 LPRINT LPOS(0)
30 LPRINT "123456789";
40 LPRINT LPOS(0)
50 END
OK

FOUR 5
123456789 10
```

EXAMPLES OF FORMATTED OUTPUT:

Example 1:

```
LIST
10 A$="DESCRIPTION"
20 B$="COST"
30 PRINT USING "\          \";A$;
40 PRINT SPC(15);:PRINT USING "\   \";B$
50 END
OK
RUN
DESCRIPTION               COST
OK
```

Example 2:

```
LIST
10 A=20
20 B=15
30 C$="THE END"
40 LPRINT TAB(15);
50 LPRINT USING "A IS ## B IS ## AND C$ IS \      \";A,B,C$
60 END
OK
```

```
A IS 20 B IS 15 AND C$ IS THE END
```

This section describes advanced ADDS*BASIC™ statements and two additional operators which will be of use to the more experienced programmer.

ADVANCED OPERATORS

Two advanced operators are provided by ADDS*BASIC™.

a. Integer Division

Integer division, denoted by \ (backslash), forces its arguments to integer form and truncates the quotient to an integer. More precisely,

```
A\B = FIX(INT(A)/INT(B))
```

Its precedence is just after multiplication and floating point division. Integer division is approximately eight times as fast as standard floating point division.

Example 1:
```
PRINT -27.1\7.9
-4
OK
```

Example 2:
```
PRINT 27.1\7.9
 3
OK
```

b. Modulus Arithmetic - the MOD operator

A MOD B forces both arguments to integer form and gives the 'remainder' as A is divided by B. More precisely,

```
A MOD B = INT(A)-(INT(B)*(A\B))
```

If B=Ø a DIVISION BY ZERO error occurs. The precedence of MOD is just below that of integer division.

Example 1:
```
PRINT -27.1 MOD 7.9
 0
OK
```

Example 2:
```
PRINT 27.1 MOD 7.9
 6
OK
```

ADVANCED STATEMENTS

The advanced statements are described on the following pages.

CLEAR

General Form: CLEAR [Expression]

expression:
any numeric expression

Purpose: To clear program variables and reserve bytes of string space equal to the value of the expression I.

Remarks: When ADDS*BASIC™ is loaded, string space is set to 100 bytes.

The CLEAR statement must prcede the DIM statement or a SUBSCRIPT OUT OF RANGE ERROR will result as the CLEAR statement releases the space allocated for the dimensioned variable.

Examples:

CLEAR	All program variables are set to zero
CLEAR 300	All program variables are set to zero and 300 bytes of string space are reserved.

DEF

General Form: DEF FNfunction name [(arguments)] = expression

function name:
a legal variable name
arguments:
one or more dummy variable names used to represent the actual arguments in the function
expression:
any numeric or string expression

Purpose: To permit the user to define functions which can be repeatedly referenced throughout a program.

Remarks: The dummy variables named in the DEF statement are not related to any variables in the program having the same name; the DEF statement simply defines the function and does not cause the execution of any calculation.

The DEF statement, like the functional operators in Section 2, produce a single result. A DEF statement must appear before the function it defines may be called. A function may be redefined by the appearance of another DEF statement with the same name.

If the arguments are omitted, the DEF statement is equivalent to an assignment statement.

Examples 1:

```
LIST
10 DEF FNA(M)=M/100    'FUNCTION A IS DEFINED
20 Y=100
30 PRINT FNA(Y)        'FUNCTION A IS USED
40 END
OK
RUN
 1
OK
```

Example 2:

```
LIST
10 REM PROGRAM TO FIND SIN X & COS X
20 REM IN 45 DEGREE INCREMENTS
30 DEF FNT(D)=D*3.14159/180
40 FOR I=0 TO 360 STEP 45
50 PRINT I,SIN(FNT(I)),COS(FNT(I))
60 NEXT
70 END
OK
RUN
 0            0             1
 45           .707106       .707107
 90           1             1.12352E-06
 135          .707108      -.707105
 180          2.62155E-06  -1
 225         -.707104      -.707109
 270         -1            -4.11958E-06
 315         -.707111       .707103
 360         -5.2431E-06    1
OK
```

In line 30 above, function T converts the measure of an angle from degrees to radians. Function T is used in line 50.

Example 3: Other examples of user defined functions

```
10  DEF FNB  (V,W)=(V+W)/2

20  DEF FND  (A,B,C)=B*B-4*A*C

30  DEF FNC$(V$,W$)=RIGHT$(V$+W$,5)
```

Line 30 returns the rightmost 5 characters of the concatenation of V$ and W$.

DEFINT/DEFSNG/DEFDBL/DEFSTR

General Form:

DEFINT	range of letters	Integer
DEFSNG	"	Single Precision
DEFDBL	"	Double Precision
DEFSTR	"	String

Purpose: To explicitly declare variable types.

Remarks: If no type definition statements are encountered, BASIC proceeds as if it had encountered a DEFSNG A-Z statement.

Examples:

15 DEFINT I-N	Variable names beginning with the letters I-N are to be of integer type.
20 DEFDBL D	Variable names beginning with D are to be of double precision type.

Note: The explicit typing of variables may be overridden by including one of the special symbols in the variable name. For example, the variable D$ refers to a string value and D% to an integer value even though statement 20 above declares all variables beginning with D to be double precision.

DEFUSR

General Form: DEFUSR [digit] = integer expression

digit:
a number from 0-9
integer expression:
starting address of the specified USR routine

Purpose: To define up to 10 assembly language subroutines.

Remarks: See Appendix D for a description of the USR function; also see PEEK and POKE.

Example:

```
100 DEFUSR1 = &100000
110 DEFUSR2 = 31096
```

IF...THEN...ELSE

General Form: IF expression THEN [expression] [line number] ELSE [expression] [line number]

expression:
any arithmetic, relational or logical expression
line number:
a program statement line number

Purpose: Same as IF...THEN statement

Remarks: A comma is allowed before THEN.

Example 1: If X>Y THEN PRINT "GREATER" ELSE PRINT "NOT GREATER"

If the expression X>Y is true, the statement after THEN is executed; otherwise, the statement after ELSE is executed.

Example 2: If X=2*Y THEN 5 ELSE PRINT "ERROR"

If expression X=2*Y is true, program control transfers to line 5; otherwise, the PRINT statement is executed.

Example 3: If the sum of variables A and B is non-zero and variable C is also non-zero, print "TRUE"; otherwise, print "FALSE".

```
LIST
10 INPUT A,B,C
20 IF A+B AND C THEN PRINT "TRUE" ELSE PRINT "FALSE"
30 GOTO 10
OK
RUN
? 2,3,4
TRUE
? 3,0,7
TRUE
? 5,6,0
FALSE
? 0,3,6
TRUE
? ^C
BREAK IN 10
OK
```

Example 4: IF statements may be nested. Nesting is limited only by the length of the logical line (maximum 255 characters). Thus, for example,

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

and

```
IF X=Y THEN IF Y>Z THEN PRINT "X>Z"
       ELSE PRINT "Y<=Z" ELSE PRINT "X<>Y"
```

are legal statements.

Example 5: If a line does not contain the same number of THEN and ELSE clauses, each ELSE is matched with the closest unmatched THEN.

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

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

INKEY

General Form: INKEY variable

variable:
numeric or string variable

Purpose: To read the keyboard directly.

Remarks: The INKEY statement will read a single key value into a variable upon depression of that key. The key is not echoed back to the screen. A numeric variable will be assigned a decimal number which is the ASCII equivalent of the key value. A string variable will be assigned the ASCII character.

Example 1: Upon depression of the Ø key and the W key, the variable X below is assigned 48 which is the decimal equivalent of ASCII Ø and 87 which is the decimal equivalent of ASCII W.

```
LIST
10 INKEY X
20 PRINT X
30 END
OK
RUN          <Depress Ø>
 48
OK
RUN          <Depress W>
 87
OK
```

Example 2: The string variable S$ below is assigned the ASCII characters Ø and W.

```
LIST
10 INKEY S$
20 PRINT S$
30 END
OK
RUN          <Depress Ø>
Ø
OK
RUN          <Depress W>
W
OK
```

For additional examples, see LOCATE.

Example 3: The string variable C$ is used to terminate the program when the user has finished entering data on the screen. A depression of the letter N permits the entry of more data. A depression of Y or any other letter ends the program.

```
LIST
10 INPUT "ENTER NAME  ";A$
20 INPUT "ENTER PHONE #";B$
30 PRINT "FINISHED(Y/N)?"
40 INKEY C$
50 IF C$="N" GOTO 10
60 END
OK
RUN
ENTER NAME   ? ADDS
ENTER PHONE #? 516-231-5400
FINISHED(Y/N)?
ENTER NAME   ? APPLIED DIGITAL DATA SYSTEMS
ENTER PHONE #? 516-231-5400
FINISHED(Y/N)?
OK
```

LOCATE

General Form: LOCATE row, column

row:
an integer constant or variable in the range 1 to 25
column:
an integer constant or variable in the range 1 to 80

Purpose: To position the cursor at a specified row and column.

Remarks: The LOCATE statement permits data to be entered or printed at a specific cursor position on the screen. This is particularly useful when creating forms.

Example 1:

```
LIST
10 LOCATE 10,8      'MOVE CURSOR TO ROW 10 COLUMN 8
20 INPUT A$         'INPUT STRING FROM R10, C8
30 LOCATE 12,30     'MOVE CURSOR TO ROW 12 COLUMN 30
40 PRINT A$         'PRINT STRING AT R12,C30
50 END
OK
RUN

       ? TESTING 1234

                                   TESTING 1234
OK
```

Example 2:

```
LIST
10 INKEY A
20 LOCATE 15,10     'MOVE CURSOR TO ROW 15 COLUMN 10
30 PRINT A          'PRINT DECIMAL VALUE
40 IF A=49 GOTO 60
50 GOTO 10
60 END
OK
RUN

          49

OK
```

Example 3:

```
LIST
10 INKEY A$
20 LOCATE 13,9    'MOVE CURSOR TO ROW 13 COLUMN 9
30 PRINT A$
40 IF A$="1" THEN GOTO 60
50 GOTO 10
60 END
OK
RUN



        1
OK
```

Note: In direct mode, the LOCATE statement is immediately followed by an automatic carriage return and the BASIC prompt Ok. This positions the cursor at column one of the row that is one or two rows below the row specified in the LOCATE statement.

POKE

General Form: POKE I,J

I,J:
integer expressions

Purpose: To enter data into memory in binary form.

Remarks: POKE stores the byte J into the location specified by the value of I. I may be in the range 0 to 65536. J must be in the range 0 to 255. Care must be taken not to POKE data into the storage area occupied by ADDS*BASIC™ or the system may be POKEd to death, and BASIC will have to be loaded again.

POKE may be used to load an assembly program into memory and to pass arguments to an assembly language program.

Note: See PEEK function below.

PEEK Function Form: PEEK(I)

The PEEK function is provided to retrieve data from memory in binary form. The argument, I, is an integer expression specifying the address from which a byte is read. I is chosen in the same way as in the POKE statement. The value returned is an integer between 0 and 255. PEEK is used to return arguments and results from assembly language programs.

SWAP

General Form: SWAP variable1, variable2

variable:
any string, numeric or array variable name

Purpose: To allow the values of two variables to be exchanged.

Remarks: The value of the second variable is assigned to the first variable and vice-versa. Either or both of the variables may be elements of arrays. Both variables must be of the same type or a TYPE MISMATCH error will result.

Example:

```
LIST
10 READ F$,L$
20 PRINT F$,L$
30 SWAP F$,L$
40 PRINT F$,L$
50 DATA "FIRST","LAST"
60 END
OK
RUN
FIRST          LAST
LAST           FIRST
OK
```

SYSTEM

General Form: SYSTEM

Purpose: To return to the **Command Processor**

Remarks: SYSTEM closes all files and then performs an ADOS warm start.

The SYSTEM statement should be included in programs run in batch mode. A return to the ADOS **Command Processor** allows the next program in the batch to be initiated.

See Appendix A - Start-up Procedures.

Example:

```
LIST
10 INPUT X
20 PRINT X
30 SYSTEM
OK
RUN
? 12
 12

A>
```

WAIT

General Form:	WAIT integer constant
Purpose:	To cause the system to wait for a specified length of time.
Remarks:	The system waits or pauses for 1/10 second times the specified integer constant before continuing.
Example 1:	WAIT 1 The system will pause for 1/10 second
Example 2:	WAIT 50 The system will pause for 5 seconds
Example 3:	WAIT 600 The system will pause for 1 minute

SECTION 8:

DEBUGGING AIDS

Numerous debugging aids are available in ADDS*BASIC™. Two statements, TRON and TROFF, are provided to trace the execution of program instructions. In addition, it is possible for the user to write error detection and handling routines which can recover from errors or provide a more complete explanation of the cause of errors than the simple error messages provided by ADDS*BASIC™. This facility has been added to ADDS*BASIC™ through the use of the ON ERROR GOTO, RESUME and ERROR statements and with the ERR and ERL variables.

Debugging is also facilitated by the use of the direct mode and the CONT statement.

TRON/TROFF

General Form: TRON

TROFF

Purpose: To trace the execution of program instructions.

Remarks: When the trace flag is turned on by the TRON statement, the number of each line in the program is printed as it is executed. The numbers appear enclosed in square brackets []. The function is disabled by execution of the TROFF statement.

TRON and TROFF may be used in both the direct mode and the indirect mode of execution (see examples).

Example 1:

```
LIST
10 A=55
20 FOR I=1 TO 2
30 B=A*I
40 PRINT A,I:PRINT B
50 NEXT
60 END
OK
TRON
OK
RUN
[10] [20] [30] [40]  55            1
 55
[50] [30] [40]  55                 2
 110
[50] [60]
OK
TROFF
OK
```

Example 2:

```
LIST
5 TRON
10 A=55
20 FOR I=1 TO 2
30 B=A*I
40 PRINT A,I:PRINT B
50 NEXT
55 TROFF
60 END
OK
RUN
[10][20][30][40] 55          1
 55
[50][30][40] 55
 110
[50][55]
OK
```

Note: The NEW statement will also turn off the trace flag.

ON ERROR GOTO

General Form: ON ERROR GOTO line number

Purpose: To enable the error trapping subroutine beginning at the specified line number.

Remarks: The ON ERROR GOTO statement should be executed before the user expects any errors to occur. Once an ON ERROR GOTO statement has been executed, all errors detected will cause BASIC to start execution of the specified error handling routine. If the line number specified in the ON ERROR GOTO statement does not exist, an UNDEFINED LINE error will occur.

ON ERROR GOTO Ø disables trapping of errors so any subsequent error will cause BASIC to print an error message and stop program execution. If an ON ERROR GOTO Ø statement appears in an error trapping subroutine, it will cause BASIC to stop and print the error message which caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO Ø subroutine if an error is encountered for which there is no recovery action.

Note: If an error occurs during the execution of an error trap routine, the system error message will be printed and execution will be terminated. Error trapping does not trap errors within the error trap routine.

Example:

```
10  ON ERROR GOTO 1000
```

This statement enables the error trapping subroutine beginning at line 1000.

See examples of error trapping subroutines on page 8-8.

THE ERR AND ERL VARIABLES

When the error handling subroutine is entered, the variable ERR contains the error code for the error. The error codes and their meanings are listed below. See Appendix G for a detailed discussion of each of the errors and error messages. Table G.2 contains a list of all disk error messages.

Code	Error
0	SYNTAX ERROR
1	NEXT WITHOUT FOR
3	RETURN WITHOUT GOSUB
4	OUT OF DATA
5	ILLEGAL FUNCTION CALL
6	OVERFLOW
7	OUT OF MEMORY
8	UNDEFINED LINE
9	SUBSCRIPT OUT OF RANGE
10	REDIMENSIONED ARRAY
11	DIVISION BY ZERO
12	ILLEGAL DIRECT
13	TYPE MISMATCH
14	OUT OF STRING SPACE
15	STRING TOO LONG
16	STRING FORMULA TOO COMPLEX
17	CAN'T CONTINUE
18	UNDEFINED USER FUNCTION
19	NO RESUME
20	MISSING OPERAND
21	RESUME WITHOUT ERROR
22	UNPRINTABLE ERROR
23	LINE BUFFER OVERFLOW
24	POSITION NOT ON SCREEN

The ERL variable contains the line number of the line where the error was detected. For instance, if the error occurred in line 1000, ERL would be equal to 1000. If the statement which caused the error was a direct mode statement, ERL would be equal to 65535 decimal. To test if an error occurred in a direct statement, use

```
IF 65535=ERL THEN ...
```

In all other cases, use

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

If the line number is on the left of the equation, it cannot be renumbered by RENUM.

Neither ERL nor ERR may appear to the left of the = sign in a LET or assignment statement.

Example 1:

```
READ A,B,C,D
OUT OF DATA
OK
PRINT ERR,ERL
 4              65535
OK
```

Example 2:

```
PRINT 4/0
DIVISION BY ZERO
OK
PRINT ERR,ERL
 11             65535
OK
```

Example 3:

```
LIST
10 A=10
20 B=23
30 PRINT A+B
40 GOTO 60
50 END
OK
RUN
 33
UNDEFINED LINE NUMBER IN 40
OK
PRINT ERR,ERL
 8              40
OK
```

RESUME

General Form: RESUME line number

RESUME NEXT

RESUME Ø

Purpose: To continue execution of the BASIC program after the error recovery procedure has been performed.

Remarks and Examples: The user may RESUME execution at the statement that caused the error,

RESUME

or RESUME Ø

at the statement immediately following the one that caused the error,

RESUME NEXT

or at some other line,

RESUME line number

See examples of error trapping subroutines on page 8-8.

EXAMPLES OF ERROR TRAPPING SUBROUTINES

The following example shows how a simple error trapping subroutine operates.

```
LIST
100 ON ERROR GOTO 200
120 INPUT "WHAT ARE THE NUMBERS TO DIVIDE";X,Y
140 Z=X/Y
160 PRINT "QUOTIENT IS ";Z
180 GOTO 120
200 IF ERR=11 AND ERL=140 THEN 240
220 END
240 PRINT "YOU CAN'T HAVE A DIVISOR OF ZERO!"
260 RESUME 100
OK
RUN
WHAT ARE THE NUMBERS TO DIVIDE? 25,5
QUOTIENT IS  5
WHAT ARE THE NUMBERS TO DIVIDE? 37,0
YOU CAN'T HAVE A DIVISOR OF ZERO!
WHAT ARE THE NUMBERS TO DIVIDE? 14,0
YOU CAN'T HAVE A DIVISOR OF ZERO!
WHAT ARE THE NUMBERS TO DIVIDE? ^C
BREAK IN 120
OK
```

The error trapping subroutine begins at line 200. If the error code equals 11 and the error occurred in line 140 the message in line 240 is printed on the screen and execution is resumed at line 100. Any other error causes the program to terminate.

In the example below, X% is an integer variable. The ON ERROR GOTO Ø statement in line 220 disables the error trapping subroutine, forcing BASIC to handle the OVERFLOW error.

```
LIST
100 ON ERROR GOTO 200
120 INPUT "WHAT ARE THE NUMBERS TO DIVIDE";X%,Y
140 Z=X%/Y
160 PRINT "QUOTIENT IS ";Z
180 GOTO 120
200 IF ERR=11 AND ERL=140 THEN 240
220 ON ERROR GOTO 0
240 PRINT "YOU CAN'T HAVE A DIVISOR OF ZERO!"
260 RESUME 100
OK
RUN
WHAT ARE THE NUMBERS TO DIVIDE? 36,6
QUOTIENT IS  6
WHAT ARE THE NUMBERS TO DIVIDE? 14,0
YOU CAN'T HAVE A DIVISOR OF ZERO!
WHAT ARE THE NUMBERS TO DIVIDE? 38000,9
OVERFLOW IN 120
OK
```

ERROR

General Form: ERROR integer expression

integer expression:
must be in the range Ø to 255

Purpose: To allow the user to define his own error codes which can then be handled conveniently by a centralized error trap routine.

Remarks: When defining error codes, values should be picked which are greater than the ones used by ADDS*BASIC™. Since more error messages may be added to ADDS*BASIC™, user-defined error codes should be assigned the highest possible numbers to assure future compatibility. If the integer expression used in an ERROR statement is less than zero or greater than 255 decimal, an ILLEGAL FUNCTION CALL error will occur. Of course, the ERROR statement may also be used to force SYNTAX or other standard ADDS*BASIC™ errors. Use of an ERROR statement to force printout of an error message for which no error text is defined will cause an UNPRINTABLE ERROR message to be printed out.

Example: In this example, the user-defined error code 77 is paired with the following error message

DIVIDEND TOO LARGE

```
LIST
10 ON ERROR GOTO 70
20 INPUT "WHAT ARE THE NUMBERS TO DIVIDE";X,Y
30 IF X>=300 THEN ERROR 77
40 Z=X/Y
50 PRINT "QUOTIENT IS: ";Z
60 GOTO 20
70 IF ERR = 11 AND ERL = 40 THEN 100
80 IF ERR = 77 THEN PRINT "DIVIDEND TOO LARGE"
90 RESUME 20
100 PRINT "YOU CAN'T HAVE A DIVISOR OF ZERO"
110 RESUME 20
OK
RUN
WHAT ARE THE NUMBERS TO DIVIDE? 333,11
DIVIDEND TOO LARGE
WHAT ARE THE NUMBERS TO DIVIDE? 16,0
YOU CAN'T HAVE A DIVISOR OF ZERO
WHAT ARE THE NUMBERS TO DIVIDE? 45,5
QUOTIENT IS:  9
WHAT ARE THE NUMBERS TO DIVIDE?  C
BREAK IN 20
OK
```

DIRECT MODE

The direct mode can be used to print the intermediate values of all the variables when execution is terminated either by a STOP statement, CTRL-C, or by an error. For example, a STOP statement may temporarily be inserted before an IF statement. At execution time, when the STOP statement is encountered, the user can print out the values of the variables to determine which branch will be taken. The program execution may then be continued with a CONT statement (see following page).

```
.
.
.
400  STOP
410  IF A+B/C>0 THEN GOTO 200
.
.
.
```

The following statement in direct mode will print out the values of the variables A,B and C.

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

Statements entered in the direct mode will be checked immediately for SYNTAX errors by the BASIC interpreter. They may be recovered by depressing CTRL-A which causes the system to enter EDIT mode (see Section 3 and Appendix A).

The direct mode is also useful when an infinite loop is suspected. An infinite loop is a series of statements from which there is no escape. Depressing CTRL-C terminates execution and enables the user to change the values of variables. Execution can be resumed by typing CONT or by entering a direct mode GOTO statement which causes execution to resume at the specified line number.

CONT

General Form: CONT

Purpose: To continue program execution after CTRL-C has been depressed, a STOP or END statement has been executed or an error has occurred.

Remarks: Execution resumes at the statement following the break unless input from the terminal was interrupted. In that case, execution resumes with the reprinting of the prompt (?) or the prompt string followed by the prompt (?).

One or more floppy disks may be connected to a System 70 and a System 50. Users with one drive should address the drive as A, and users with two drives should address them as A and B, etc.

Important: The RESET statement must be entered immediately after switching disks in any drive. RESET constructs a new map of the directory.

FILE NAMES

BASIC creates both program files and data files. All data and program files on the disk have associated file names. The form of a program or data file name is:

filename.filetype

Each filename contains a maximum of eight characters and each filetype contains up to three characters. Although the filetype is optional, it is usually generic. For example, the filetype DAT is used to denote a data file and BAS to denote a BASIC program file. A filename over eight characters in length will cause a BAD FILE NAME error.

The BASIC statements to load, save, list, merge, delete, rename and open files require a file reference. A file reference can be specific or general. The specific file reference PGM1.BAS uniquely identifies a single file while the general file reference *.BAS may be satisfied by a number of different files. See Section 3 of the ADOS Manual for a complete discussion of specific and general file references.

The following abbreviations are used in the formats of the statements in this section.

drv: disk drive - if omitted drive A is assumed.
sfn: specific file reference
gfn: general file reference

The file reference used with the BASIC statements LOAD, SAVE, KILL, FILES and OPEN may be contained in a string variable name (see SAVE, OPEN statements).

GENERAL DISK FILE STATEMENTS

The following three ADDS*BASIC™ statements are used to list, rename and delete both program and data files.

FILES

General Form: FILES ["[drv:]gfn"]

drv: disk drive -- if omitted, drive A is assumed
gfn: general file reference

Purpose: To list the names of all files satisfying the given general file reference.

Remarks: Any time a general file reference is called for, a specific file reference may also be used.

Examples:

FILES	lists all the files on disk drive A
FILES "*.DAT"	lists all the data files (files with filetype DAT) on disk drive A
FILES "B:*.*"	lists all the files on disk drive B
FILES "B.PGM?.BAS"	lists all the BASIC program files having filetype BAS on disk drive B and satisfying the general file reference PGM?-- PGM1.BAS, PGM2.BAS, etc. satisfy this general file reference.
FILES "MYFILE.DAT"	lists the single file having the specific file reference MYFILE.DAT on drive A if it exists.

NAME

General Form: NAME "[drv:]old sfn" AS "[drv:]new sfn"

drv: disk drive -- if omitted, drive A is assumed
sfn: specific file reference

Purpose: To change the name of a file.

Remarks: The old filename must exist, or a FILE NOT FOUND error will occur. A file with the same name as the new filename must not exist or a FILE ALREADY EXISTS error will occur. After the NAME statement is executed, the file exists on the same disk in the same area of disk space. Only the name is changed.

Example:

```
NAME "B:PGM1.BAS"  AS  "B:PGMA.BAS"
```

PGM1.BAS on disk drive B is renamed PGMA.BAS.

KILL

General Form: KILL "[drv:]sfn"

drv: disk drive -- if omitted, drive A is assumed
sfn: specific file reference

Purpose: To delete a file from disk and return disk space used by the file to free disk space.

Remarks: If the file does not exist, a FILE NOT FOUND error will occur.

Example 1: KILL "ACCT.DAT" Deletes data file ACCT.DAT on drive A.

Example 2: KILL "B:PGMA.BAS" Deletes program file PGMA.BAS on disk drive B.

PROGRAM FILES

Once a program has been written, it is often desirable to save it on disk for use at a later time. Unless otherwise specified, ADDS*BASIC™ saves its programs in a compressed binary format which takes a minimum of disk space and loads very quickly. The user has the option of saving a program in ASCII, a form that allows program text to be read as data by another program, such as a text or line editor.

When a program file, previously saved on disk, is loaded into memory, it may be executed automatically or edited, executed and saved again on disk. If required, it is also possible to merge a program file stored on disk with a program already in memory.

SAVE

General Form: SAVE "[drv:]sfn"[,A]

drv: disk drive; if omitted, drive A is assumed
sfn: specific filename
A: ASCII option (see Remarks below)

Purpose: If the filetype is omitted, it is assumed to be BAS. If the "A" option is present, the file is saved in ASCII format. The user then has no way of determining if the program file is in ASCII or binary format. The user may choose descriptive extensions such as BNY or ASC for binary and ASCII formats.

Whenever a program is SAVEd, any existing copy of the program previously SAVEd will be deleted, and the disk space used by the previous program is made available.

Example 1:

```
SAVE "PGM1"
```

saves file PGM1.BAS in binary format on disk drive A.

Example 2:

```
SAVE "B:ACCTS.BAS",A
```

saves file ACCTS.BAS in ASCII format on disk drive B.

Example 3:

```
X$="MYFILE"
SAVE X$
```

saves file MYFILE.BAS in compressed binary format on disk drive A.

LOAD

General Form: LOAD "[drv:]sfn"[,R]

drv: disk drive; if omitted, it is assumed to be drive A
sfn: specific file name
R: execute option

Purpose: To read a file from disk and load it into memory.

Remarks: If the filetype is omitted it is assumed to be BAS. Information in the file tells the LOAD command the format in which the file is to be loaded. Remember, loading an ASCII file is much slower than loading a binary file. If a file has been saved in compressed binary format, it may be loaded and then SAVEd again in ASCII format using either the same file name or a different file name. If the file does not exist, a FILE NOT FOUND error will occur.

The LOAD command with the "R" option is used to chain segmented programs when the whole program is too large to fit in the computer's memory. All variables and program lines in memory are deleted by LOAD, but all data files are kept OPEN if the "R" option is used. Therefore, information may be passed between programs through the use of disk data files. If the "R" option is not used, all files are automatically CLOSEd by a LOAD statement.

Example 1:

```
LOAD "PGM1"
OK
LIST
10 PRINT "PGM1":LOAD "PGM2",R
OK
LOAD "PGM2"
OK
LIST
10 PRINT "PGM2":LOAD "PGM1",R
OK
RUN
PGM2
PGM1
PGM2
PGM1
PGM2
^C
BREAK IN 10
OK
```

(CTRL-C may be used to stop execution at this point.) In this example, program PGM2 is RUN. PGM2 prints the message "PGM2" and then loads (from drive A) and executes the program PGM1. PGM1 prints "PGM1" and calls the program PGM2 which prints "PGM2" and so on indefinitely. Also see RUN command.

Example 2:

```
LOAD "LITEST"
OK
LIST
10 OPEN "0",1,"TEST.DAT"
20 INPUT A,B,C
30 PRINT #1,A,B,C
40 PRINT A,B,C
50 LOAD "L2TEST",R
OK
RUN
? 1,2,3
 1            2            3
THE END
OK
LIST
10 X$="THE END"
20 PRINT #1,X$
30 CLOSE
40 PRINT X$
OK
```

In the above example, the R option is used in line 50 to keep the file TEST.DAT OPEN during execution of program L2TEST. (See OPEN statement.)

RUN

General Form: RUN [line number]
RUN "[drv:]sfn" [,R]

line number:	a program line number
drv:	disk drive; if omitted, drive A is assumed
sfn:	specific file name
R:	OPEN files option - see Remarks below

Purpose: To execute a program.

Remarks: The statement

```
RUN  [line number]
```

starts execution of the program currently in memory at the line specified. If the line number is omitted, the entire program is executed.

The statement

```
RUN  "[drv:]sfn"
```

starts execution of the specified filename on the specified disk drive. If the disk drive is omitted, it is assumed to be A.

All files are closed unless ",R" is specified after the file name. (See example below.)

Example: The program MYPROG listed below is used in this example. The file TEST.DAT is OPENed in direct mode before the RUN statement with the R option is used. Notice the FILE ALREADY OPEN error.

```
LIST
10 OPEN "0",1,"TEST.DAT"
20 INPUT A,B,C
30 PRINT #1,A,B,C
40 PRINT A,B,C
OK

OPEN "0",#1,"TEST.DAT"
OK
RUN "MYPROG",R
FILE ALREADY OPEN IN 10
OK
```

MERGE

General Form: MERGE "[drv:]sfn"

drv: disk drive; if omitted, drive A is assumed
sfn: specific file name

Purpose: To merge parts of two programs together to form a new program combining elements of both programs.

Remarks: The file name specified is merged into the program already in memory. The file name must specify a program SAVEd in ASCII format or a BAD FILE MODE error will occur. If there are lines in the program on disk which have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding program lines in memory. It is as if the program lines of the file on disk were typed on the user terminal.

If the file type is omitted, it is assumed to be BAS.

Example:

```
MERGE  "PGM2.ASC"
```

The above command merges PGM2.ASC into the program stored in memory.

DATA FILES

Techniques have been presented for entering data into a program as the program is written (via assignment, READ and DATA statements) or from the keyboard while the program is executing (via INPUT statements). Both techniques are inefficient when the amount of data to be read or written increases beyond a few items. In order to improve operation, ADDS*BASIC™ provides the user with facilities to define and manipulate Input/Output disk data files.

ADDS*BASIC™ supports Sequential Access, Random Access and ADDS*PLUS™ data files. A programmer who wishes to read and write data to a disk file must first OPEN the file on the appropriate disk drive. Up to eight files may be opened simultaneously.

GENERAL RULES

1. one or more disk drives are supported.
2. Up to 8 files may be OPEN simultaneously.
3. All disk access operations perform retry error recovery procedures three times. For a write that fails when writing, retries are repeated as with read.
4. A particular file name (filename.filetype) must be unique on any one drive but may exist on other drives. Therefore, file name and drive specify a unique file to the system.

Each file being OPENed is associated with a file number, an integer between one and eight. This file number is used to refer to the file in later I/O operations. A file number greater than three may not be used unless the F option was specified at the time the BASIC Interpreter was invoked (see Appendix A). After I/O operations are completed, each OPENed file must be CLOSEd. The formats of the OPEN and CLOSE statements (which are identical for all three file types) follow.

OPEN

General Form: OPEN "mode", [#]file #,"[drv:]sfn"

mode	meaning
O	Sequential Output mode
I	Sequential Input mode
R	Random Input/output mode
D	ADDS*PLUS™ Input/Output mode

The mode may also be a string expression whose first character is one of the above.

file number:
an integer expression between 1 and 8. The number is associated with the file being OPENed and is used to refer to the file in later I/O operations.

drv: disk drive; if omitted, drive A is assumed

sfn: specific file name

Purpose: To open a Sequential, Random or ADDS*PLUS™ data file for reading and/or writing.

Remarks: If the file type is omitted, it is assumed to be three blanks.

Up to eight files may be OPENed simultaneously. A FILE NOT FOUND error results if the file does not exist or has been specified incorrectly. A FILE ALREADY OPEN error occurs if the file was OPENed previously.

When an OPEN statement has been successfully executed, the pointer is positioned at the beginning of the file.

If no diskette has been placed in the specified drive with ADOS Release 1.4 and later, a

```
READY DRIVE n
```

error occurs. Another possible error

```
PERM DISK ERROR
```

is unrecoverable, and indicates that several retries were made to read the track. This sequence is performed 3 times.

Example 1:

```
OPEN "O",2,"OUTPUT"
OPEN "I",1,"INPUT"
```

The above two statements will open the file OUTPUT for Sequential Output and the file INPUT for Sequential Input on disk A.

Example 2:

```
OPEN "R",6,"RAN"
OPEN "D",8,"B:APLUS"
```

The above two statements will open the file RAN on drive A for Random I/O and the file APLUS on drive B for ADDS*PLUS™ I/O.

Example 3:

```
OPEN  M$,N,F$
```

M$	mode
N	file number
F$	[drv:] sfn

The above statement would open the file whose name was in the string F$ in mode M$ as file number N on the specified drive.

CLOSE

General Form: CLOSE [file number[,file number ...]]

file number:
represents the number of the file that was OPENed for output (must be an integer from 1-8)

Purpose: To close a data file that has been opened for I/O and release the file number for use by other files.

Remarks: After CLOSE has been executed for a file, the file may be reOPENed for input or output on the same or a different file number. A CLOSE for a Sequential Output file writes the final buffer of output. A CLOSE to any OPEN file terminates the connection between the specified file number and the file name used in the OPEN for that file. It allows the specified file number to be used again in another OPEN statement.

A CLOSE with no argument CLOSEs all OPEN files.

Note: A file can be OPENed for Sequential Input or Random Access on more than one specified file number at a time but may be OPENed for output on only one file number at a time.

END and NEW always CLOSE all disk files automatically. STOP does not CLOSE disk files.

Example: See INPUT # Statement.

SEQUENTIAL ACCESS FILES

A Sequential Access file is a stream of characters that is read or written in order, much like INPUT and PRINT statements read data from and write data to the screen. Sequential I/O is the simplest form of disk I/O operation since it involves the use of INPUT # and PRINT # statements with a file that has been previously OPENed. (See OPEN statement.)

All data items must be delimited with the comma or ENTER key.

A Sequential Access file may be OPENed for input on one file number and for output on another file number.

CAUTION

> If data is output to an existing Sequential Access file, the entire file will be deleted and replaced with a new file of the same name. In order to add records to a Sequential Access file, the file must be opened for input with a second Sequential Access file opened for output. Each record input may then be written to the new output file and the additional records output after the records on the original file have been exhausted. Records may be updated in a similar fashion.

INPUT

General Form: INPUT # file number, variable list

file number:
represents the number of the file that was OPENed for input (must be an integer from 1-8)
variable list:
list of the variables to be read as in a normal INPUT statement

Purpose: To read data from a Sequential Access file.

Remarks: When data is read from a Sequential Access file using an INPUT # statement, no question mark (?) is printed on the terminal. The format of data in the file should appear exactly as it would be typed in response to a standard INPUT statement to the terminal.

When reading numeric values, leading spaces and ENTER key characters are ignored. When any other character is found, it is assumed to be part of a number in ADDS*BASIC™ format. The number terminates on a space, comma or ENTER key character.

When scanning for string items, leading blanks, ENTER and NEW LINE key characters are also ignored. When any other character is found, it is assumed to be the start of a string item. If this first character is a quotation mark (") the item is taken as being a quoted string, and all characters between the first double quote (") and a matching double quote are returned as characters in the string value. This means that a quoted string in a file may contain any characters except double quote. If the first character of a string item is not a quotation mark, then it is assumed to be an unquoted string constant. The string returned will terminate on a comma, or ENTER key character. The string is immediately terminated after 255 characters have been read.

For both numeric and string items, if end of file (EOF) is reached when the item is being INPUT, the item is terminated regardless of whether or not a closing quote was encountered.

Sequential I/O commands destroy the input buffer so they may not be edited by CTRL-A for re-execution. (See Section 3 and Appendix A).

Example:

```
500  OPEN "O",1,"MYFIL1"
510  PRINT #1,X,Y,Z
520  CLOSE 1
530  OPEN "I",1,"MYFIL1"
540  INPUT #1,X,Y,Z
550  CLOSE  1
```

Note:

CLOSE is used so that a file which has just been written may be read. When MYFIL1 is reOPENed, the data pointer for that file is set back to the beginning of the file so that the first INPUT # statement will read data from the start of the file.

PRINT # and PRINT # USING

General Form: PRINT # file number, print list
PRINT # file number, USING "format string";print list

file number:
represents the number of the file that was OPENed for output (must be an integer from 1-8)
format string:
string variable, expression or constant which is a precise copy of the line to be printed
print list:
list of expressions to be printed separated by semi-colons (;) or commas (,)

Purpose: To write data to a Sequential Access file.

Remarks: Recall that all data items must be delimited by one of the following characters:

ENTER key
comma

Example 1: When leading blanks are not significant and there are no commas or ENTER key characters in the strings to be output, commas may be inserted between the strings being output as follows:

```
LIST
470 X$="ADDS"
480 Y$="SYSTEM"
490 Z$="50/70"
500 OPEN "O",1,"MYFIL1"
510 PRINT #1,X$;",";Y$;",";Z$
520 CLOSE 1
530 OPEN "I",1,"MYFIL1"
540 INPUT #1,X$,Y$,Z$
550 PRINT X$,Y$,Z$
560 END
OK
RUN
ADDS          SYSTEM       50/70
OK
```

The commas enclosed in quotation marks serve as delimiters of the strings X$, Y$ and Z$ in statement 510. Notice that the filename "MYFIL1" is closed in statement 520 before it is OPENed for input.

See INPUT # statement for an example of output of numeric data to a Sequential Access file.

Example 2:

If a string which is being output to a Sequential Access file contains commas or ENTER keys or leading blanks that are significant, it must be surrounded with double quotes through the use of the CHR$ function to generate the ASCII value for a double quote. CHR$(34) will generate a double quote.

```
LIST
10 OPEN "O",1,"MYPROG"
20 X$="    HAUPPAUGE, NEW YORK"
30 PRINT #1,CHR$(34);X$;CHR$(34)
40 PRINT #1,CHR$(34);"1,000,000";CHR$(34)
50 CLOSE 1
60 OPEN "I",1,"MYPROG"
70 INPUT #1,X$,Y$
80 PRINT X$,Y$
90 CLOSE 1
100 END
OK
RUN
    HAUPPAUGE, NEW YORK     1,000,000
OK
```

In statements 30 and 40 the string literal "1,000,000" and string variable, X$ are surrounded with double quotes. Note the comma and leading blanks in X$.

Notice again that the Sequential Access file "MYFIL1" is CLOSEd for output before it is OPENed for input.

LINE INPUT

General Form: LINE INPUT # file number, string variable name

file number:
represents the number of the file that was OPENed for input (must be an integer from 1-8)

Purpose: To read a whole line of a file into a string without using quotes, commas or other characters as delimiters.

Remarks: This statement is useful if certain fields of each line are being used to contain data items, or if a BASIC program saved in ASCII mode is being read as data by another program.

A LINE INPUT # from a data file will return all characters up to the end of the logical line (ENTER key was depressed) in the specified string variable. LINE INPUT # then skips over the following ENTER/NEW LINE sequence so that a subsequent LINE INPUT # from the file will return the next line.

Example: See EOF function.

EOF Function Form: EOF (file #)

When reading a Sequential Access file with INPUT statements it is usually desirable to detect when there is no more data in the disk file. The EOF function is the mechanism for detecting this condition.

EOF returns TRUE (-1) when there is no more data in the file or if the file does not exist and FALSE (Ø) otherwise. If an attempt is made to INPUT past the end of a data file, an INPUT PAST END error will occur.

Example:

```
LIST
10 OPEN "O",2,"DATA3"
20 INPUT X$
30 PRINT #2,X$
40 INPUT "DO YOU WISH TO ENTER MORE DATA";D$
50 IF D$="Y" GOTO 20
60 CLOSE 2
70 OPEN "I",2,"DATA3"
80 IF EOF(2) THEN 120
90 INPUT #2,A$
100 PRINT A$,
110 GOTO 80
120 CLOSE 2
OK
RUN
? ADDS
DO YOU WISH TO ENTER MORE DATA? Y
? SYSTEM
DO YOU WISH TO ENTER MORE DATA? Y
? 50/70
DO YOU WISH TO ENTER MORE DATA? N
ADDS          SYSTEM        50/70
OK
```

In statement 90 string data from the Sequential Access file DATA3 is read into the string variable A$. When end of file is detected, the IF statement at line 80 branches to line 120 and the file is closed.

The following is an example of a program that will calculate the number of lines in a BASIC program file that has been SAVEd in ASCII mode.

Example:

```
LIST
10 INPUT "WHAT IS THE NAME OF THE PROGRAM";P$
20 OPEN "I",3,P$
30 I=0
40 IF EOF(3) THEN 70
50 I=I+1:LINE INPUT #3,L$
60 GOTO 40
70 PRINT "PROGRAM ";P$;" IS ";I;" LINES LONG"
80 END
OK
RUN
WHAT IS THE NAME OF THE PROGRAM? "ADDSPLUS.BAS"
PROGRAM ADDSPLUS.BAS IS  58  LINES LONG
OK
```

This example uses the LINE INPUT statement to read each line of the program into the "dummy" string L$ which is then ignored.

LOC Function Form: LOC(file #)

The LOC function returns the number of sectors of the Sequential Access file that have been read or written since the OPEN statement was executed. Use of the LOC function after the file is closed results in a BAD FILE NUMBER error.

128 bytes = 1 sector

Example:

```
LIST
10 INPUT "WHAT IS THE NAME OF THE PROGRAM";P$
20 OPEN "I",3,P$
30 I=0
40 IF EOF(3) THEN 70
50 I=I+1:LINE INPUT #3,L$
60 GOTO 40
70 PRINT "PROGRAM ";P$;" IS ";I;" LINES LONG"
75 PRINT LOC(3)
80 END
OK
RUN
WHAT IS THE NAME OF THE PROGRAM? "ADDSPLUS.BAS"
PROGRAM ADDSPLUS.BAS IS  58  LINES LONG
 15
OK
```

Notice the use of the LOC function in line 15.

INPUT$ Function Form: INPUT$(X[,[#]file#])

The INPUT$ function returns a string of X characters read in from a Sequential Access file. If X is too large, an INPUT PAST END error will occur. The examples below all read the file MYFIL1.BAS created by the program labeled example 1 under the PRINT # statement.

Example 1: The INPUT$ function in statement 30 reads 17 characters from the data file

```
LIST
10 OPEN "I",1,"MYFIL1"
20 PRINT
30 Y$=INPUT$(17,1)
40 PRINT "OUTPUT",,Y$
50 CLOSE 1
60 PRINT
70 END
OK
RUN

OUTPUT                          ADDS,SYSTEM,50/70

OK
```

Example 2:

```
LIST
10 OPEN "I",1,"MYFIL1"
20 PRINT
30 Y$=INPUT$(9,1)
40 PRINT "OUTPUT",,Y$
50 Y$=INPUT$(6,1)
60 PRINT "OUTPUT",,Y$
70 CLOSE 1
80 PRINT
90 END
OK
RUN

OUTPUT                          ADDS,SYST
OUTPUT                          EM,50/

OK
```

The first INPUT$ function, after the OPEN statement, reads the first nine characters from the data file. The next INPUT$ function, statement 50, reads the next six characters from the file. To read characters from the beginning of the file, it must be closed and reopened before each INPUT$ function is executed (see Example 3).

Example 3: The INPUT$ function in statement 40 always reads in characters from the beginning of the file. If the number of characters to be read in is larger than those remaining in the file, an INPUT PAST END error will occur.

```
LIST
10 OPEN "I",1,"MYFIL1"
20 INPUT "ENTER # CHAR. TO BE READ";I
30 PRINT
40 Y$=INPUT$(I,1)
50 PRINT "OUTPUT",,Y$
60 CLOSE 1
70 PRINT
80 GOTO 10
OK
RUN
ENTER # CHAR. TO BE READ? 9

OUTPUT                          ADDS,SYST

ENTER # CHAR. TO BE READ? 17

OUTPUT                          ADDS,SYSTEM,50/70

ENTER # CHAR. TO BE READ? 20

INPUT PAST END IN 40
OK
```

If the file number is omitted, INPUT$(X) returns a string of X characters input from the screen during program execution (see Section 5).

RANDOM ACCESS FILES

It is often desirable to access data in a random fashion, for instance, to retrieve information on a particular part number or customer from a large data base stored on a floppy disk. If Sequential Access files were used, the whole file would have to be scanned from the start until the particular item was found. Random Access files remove this restriction and allow a program to access any record from the first to the last in a speedy fashion.

In addition, Random Access files may be updated and added to without re-writing the entire file. Output of a record to a Random Access file re-writes the record, if it exists. If not, the file is extended to include the newly written record. (See GET statement CAUTION.)

Also, Random Access files transfer data from variables to the disk output records and vice versa in a much faster, more efficient fashion than Sequential files. Random file I/O is more complex than Sequential I/O, and it is recommended that beginners try Sequential I/O first.

Opening a Random Access File

Random I/O files are OPENed just like Sequential files.

```
OPEN  "R",1,"RANDOM"
```

When a file is OPENed for Random I/O, it is always OPEN for both input and output simultaneously.

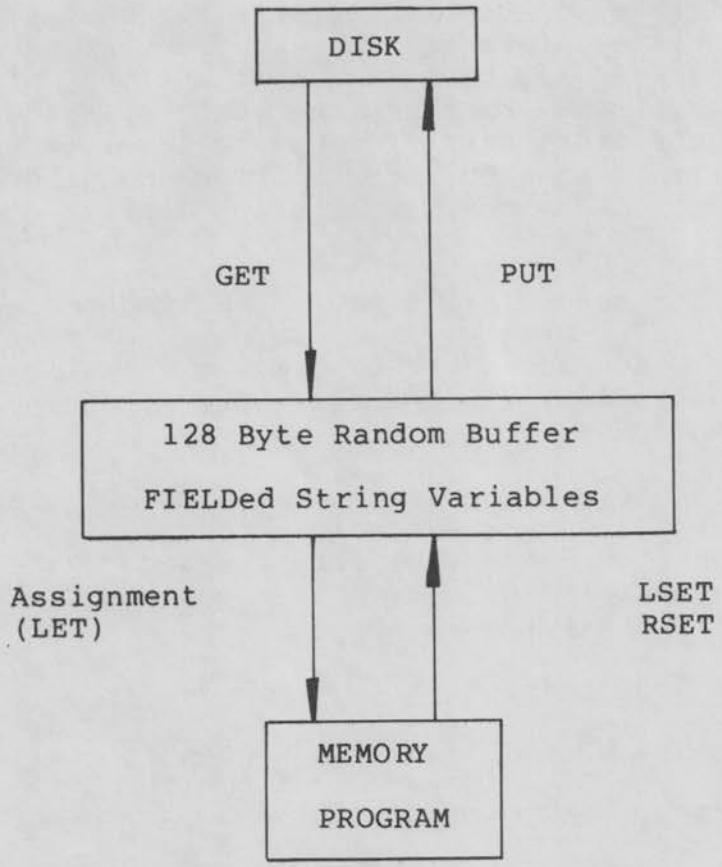
Closing Random Files

Like Sequential files, Random files must be CLOSEd when I/O operations are completed. To CLOSE a Random file, use the CLOSE statement as described previously.

```
CLOSE [[file number],file number...]
```

Random File I/O

Each Random file has associated with it a "Random Buffer" of 128 bytes or characters. Data to be written to a Random file must be moved to this buffer before it can be transferred to the data file. Similarly, data must be transferred to the buffer from the Random file before it can be input to the program. Data from string variables is moved from memory to the Random Buffer through the use of the LSET and RSET statements. Moving data from the buffer into memory is accomplished by assignment (LET) statements. The GET and PUT statements transfer data between the Random Buffer and the disk, and the FIELD statement points to the area in the string buffer where each string variable will be located. Two functions, LOC and LOF, determine the current record number and the number of records present in the current extent. These are described on the subsequent pages.



To write data to a Random file, the disk I/O statement would be used in the following order:

OPEN
FIELD
LSET or RSET
PUT
CLOSE

To read data from a Random file the following order should be followed:

OPEN
FIELD
GET
Assignment (LET)
CLOSE

Numeric/String Conversion Functions

Data is always stored in the Random Buffer through the use of string variables. A numeric value must be converted to a specially packed string before before being stored in the Random Buffer. A number of functions have been provided to convert numbers to their packed string representtions for output and to convert these packed string representations back to their numeric values for input.

To convert numbers to their packed string representation:

MKI$(integer value)	Returns a two byte string (an ILLEGAL FUNCTION CALL error results if value is not in the range -32768 to +32767). (fractional part is lost)
MKS$(single precision value)	Returns a four byte string
MKD$(double precision value)	Returns an eight byte string

To convert packed strings to numbers:

CVI(two byte string)	Returns an integer value
CVS(four byte string)	Returns a single precision value
CVD(eight byte string)	Returns a double precision value

CVI, CVS and CVD all give an ILLEGAL FUNCTION CALL error if the string given as the argument is shorter than required. If the string argument is longer than necessary, the extra characters are ignored.

These functions are extremely fast, since they convert between ADDS*BASIC™'s internal representations of integers, single and double precision values and strings. Conventional Sequential I/O must perform time-consuming character scanning algorithms when converting numbers to strings.

These functions must not be confused with the string functions VAL and STR$. The STR$ function converts a number to its string representation and the VAL function converts the string to its numeric value. However, this is not the packed string representation required for Random Access files.

```
LIST
10 PRINT "ORIGINAL","STR$ & VAL","MKS$ & CVS"
20 PRINT "VALUE","CONVERSION","CONVERSION"
30 PRINT
40 N=-12.34
50 A$=STR$(N)
60 B$=MKS$(N)
70 PRINT N,A$,B$,"NUMERIC TO STRING"
80 X$="98.76"
90 Y=VAL(X$)
100 PRINT TAB(1);X$,Y,Z,"STRING TO NUMERIC"
OK
RUN
ORIGINAL       STR$ & VAL     MKS$ & CVS
VALUE          CONVERSION     CONVERSION

-12.34         -12.34         $P             NUMERIC TO STRING
 98.76          98.76          7.20556E-23   STRING TO NUMERIC
OK
```

See LSET/RSET, PUT and GET statements for additional examples of these functions.

FIELD

General Form: FIELD [#] file number, field size AS string variable[,...]

file number:
specifies the file number of the file whose Random Buffer is being referenced (must be an integer from 1-8)

field size:
length of the string in the Random Buffer (must be in the range 1 to 128)

string variable:
string variable name which is to be associated with a specified number of characters (field size) in the buffer

Purpose: To associate some or all of a file's Random Buffer with particular string variables.

Remarks: A space must be present before and after the word "AS" in the FIELD statement.

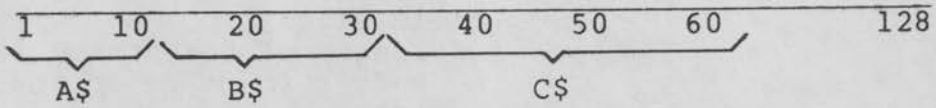
If the file is not a Random file, a BAD FILE MODE error will occur.

The FIELD statement may be executed any number of times on a given file. It does not cause any allocation of string space. The only space allocation that occurs is for the string variables mentioned in the FIELD statement. These string variables have a one byte count and two byte pointer set up which points into the Random Buffer for the specified file.

Example 1: Multiple fields may be associated with string variables in a given FIELD statement. Each successive string variable is assigned a successive field in the Random Buffer.

```
FIELD,#1,10 AS A$, 20 AS B$, 30 AS C$
```

The statement above assigns the first 10 characters of the Random Buffer to the string variable A$, the next 20 characters to B$ and the next 30 characters to the variable C$.



It is important to note that the FIELD statement does not cause any data to be transferred to or from the Random Buffer. It only causes the string variables given as arguments to "point" into the Random Buffer.

Example 2: Often it is necessary to divide the Random Buffer into a number of sub-records to make more efficient use of disk space. For instance, it might be desirable to divide the 128 character record into two identical sub-records. To accomplish this, a "dummy variable" would be placed in the FIELD statement to represent one of the sub-records. One of the following statements would be executed depending on whether the first or second sub-record were needed.

```
FIELD #1,64 AS D$, 20 AS N$,
         20 AS A$, 24 AS O$
```

or

```
FIELD #1,20 AS N$, 20 AS A$,
         24 AS O$, 64 AS D$
```

where the dummy variable D$ is used to skip over one of the sub-records.

Another way to accomplish the same thing would be to use a variable I to select the first or second sub-record.

```
FIELD #1,64*(I-1) AS D$,
         20 AS N$, 20 AS A$, 24 AS O$
```

Here, if the variable I is one, (I-1)*64 = 0 characters will be skipped over, selecting the first sub-record. If I is two, 64 characters will be skipped over, selecting the second sub-record.

Another technique that is very useful is to use a FOR...NEXT loop and an array to set up sub-records in the Random Buffer:

```
1000 FOR I=1 to 16
1010 FIELD  #1, (I-1)*8 AS D$, 4 AS A$(I),
     4 AS B$(I)
1020 NEXT I
```

In the example above, the Random Buffer is divided into 16 sub-records composed of two fields each. When I=1, the pointer is positioned at the beginning of the buffer and the first four characters are assigned to A$(1); the second four characters to B$(1). D$ is a dummy variable used to offset the pointer the required number of positions. When I=2, the pointer is offset eight characters and characters 9 through 12 are assigned to A$(2); 13 through 16 to B$(2), etc.

The same results could be achieved with the following FIELD statement:

```
1010 FIELD #1, 4 AS A$(1),4 AS B$(1), 4 AS
             A$(2), 4 AS B$(2),...,4 AS A$(16),
             4 AS B$(16)
```

Example 3: A single field may be associated with a string variable in a given FIELD statement and the data in this variable may be separated with the LEFT$, MID$ and RIGHT$ string functions.

```
LIST
10 OPEN "R",1,"MYFILE"
20 FIELD #1, 35 AS A$
30 LSET A$="APPLIED DIGITAL DATA SYSTEMS"
40 PUT #1,4
50 FIELD #1, 35 AS B$
60 GET #1,4
62 X$=LEFT$(B$,1)
64 Y$=MID$(B$,9,1)
66 Z$=MID$(B$,17,1)
68 W$=MID$(B$,22,1)
70 PRINT X$,Y$,Z$,W$
80 CLOSE 2
OK
RUN
A             D             D             S
OK
```

LSET/RSET

General Form: LSET string variable = string expression

RSET string variable = string expression

string variable:
string variable name previously mentioned in FIELD statement
string expression:
combination of string literals, variables and functions

Purpose: To move data from memory to the Random Buffer. LSET left-justifies the string, adding blanks (octal 40, decimal 32) to pad out the right side of the string if it is too short. RSET right-justifies the string, padding on the left. If the string value is too long, the extra characters at the end of the string are ignored.

Remarks: LSET and RSET may be used to assign values right- or left-justified to any string variable.

When a GET operation is performed, all string variables which have been FIELDed into the Random Buffer for that file automatically have values assigned to them. The CVI, CVS and CVD functions may be used to convert any string fields in the record to their numeric values. When going the other way, i.e., inserting strings into the Random Buffer before performing a PUT statement, a problem arises. This is because of the way string assignments usually take place.

When an assignment (LET) statement is executed,

```
[LET] A$="COST"
```

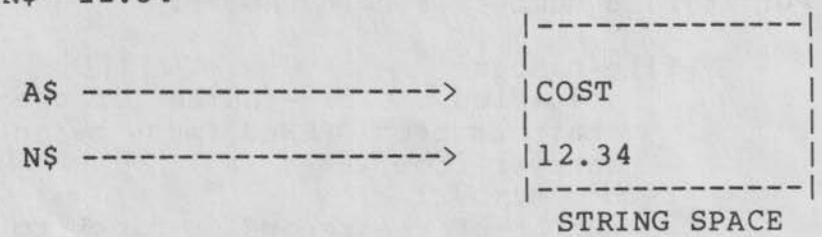
"COST" is copied into string space, A$ is pointed to the new string and the string length of A$ is modified. However, for assignments into the Random Buffer we do not want this to happen. Instead, we want the string being assigned to be stored where the string variable was FIELDed.

The LSET and RSET assignment statements accomplish this purpose (see diagrams on the following page).

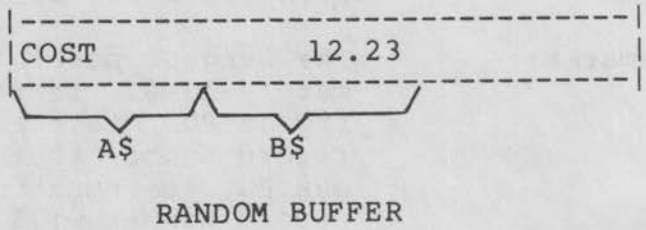
STATEMENT

STRING ASSIGNMENT

```
[LET] A$="COST"
[LET] N$="12.34"
```




```
FIELD 10 AS A$, 10 AS B$
LSET A$="COST"
RSET B$=MKS$(12.34)
```



PUT

General Form: PUT [#]file number[,record number]

file number:
specifies the file number of the file that has been OPENed (must be an integer from 1-8)

record number:
number of the record or block to be written

Purpose: To transfer data from the Random Buffer to the disk (to write data on disk).

Remarks: The largest possible record number is approximately 1900. If the record number is omitted from a PUT statement, the data is written to the record number that is one higher than the previous PUT statement. Initially, a PUT without a record number will write the first record.

Example:

```
LIST
10 OPEN "R",1,"ACCT.DAT"
20 FIELD #1, 2 AS A$, 4 AS B$, 8 AS C$, 10 AS D$
30 RSET A$=MKI$(123)
40 RSET B$=MKS$(43.99)
50 RSET C$=MKD$(123456789#)
60 LSET D$="TESTING"
70 PUT 1,5
80 CLOSE
OK
RUN
OK
```

The integer, single precision and double precision constants in lines 30, 40 and 50 are converted to their packed string representations and stored right-justified in the Random Buffer.

The string constant "TESTING" is left-justified before being stored in the Random Buffer.

Example 1 on the following page reads in this data and displays it on the screen.

GET

General Form: GET [#]file number[,record number]

file number:
specifies the file number of the file that has been opened (must be an integer from 1-8)

record number:
number of the record or block to be read

Purpose: To transfer data from the disk to the Random Buffer (to read data from disk)

Remarks: The largest possible record number is approximately 1900.

If the record number is omitted from a GET statement, the record number that is one higher than the previous GET is read into the Random Buffer. Initially, a GET without a record number will read the first record.

CAUTION

If an attempt is made to GET a record which has never been PUT, the system may retrieve invalid data. The statement PUT #1,300 does not cause the system to allocate space for the entire file (300 records). A PUT statement (mod 8) reserves space for one logical block, where:

1 logical block = 8 sectors (records)
1 sector (record) = 128 bytes

Some intermediate records may not have disk space allocated. A GET of one of these intermediate records may retrieve data from a file that has been deleted from the diskette by the ADOS command ERA or the BASIC statement KILL. When a file is deleted, the filename is removed from the directory and the previously allocated space is assigned to the free list.

Also, an attempt to use the PIP Utility to copy a Random Access file all of whose records do not have space allocated will abort upon encountering the first unallocated block.

It is recommended that all Random Access files be created in a sequential manner to maximum-use size so that space will be allocated for all blocks.

Example 1:

```
LIST
10 OPEN "R",1,"ACCT.DAT"
20 FIELD #1, 2 AS A$, 4 AS B$, 8 AS C$, 10 AS D$
30 GET 1,5
40 K%=CVI(A$)
50 L!=CVS(B$)
60 M#=CVD(C$)
70 PRINT K%,L!,M#,D$
80 CLOSE
OK
RUN
 123          43.99          123456789   TESTING
OK
```

Example 2:

The same FIELD statement may suffice for both input and output.

```
LIST
10 OPEN "R",1,"MYFILE"
20 FIELD #1, 35 AS A$
30 FOR I=3 TO 5
40 INPUT "ENTER THE NAME OF A PRESIDENT";X$
50 LSET A$=X$
60 PUT 1,I
70 NEXT I
80 FOR I=3 TO 5
90 GET 1,I
100 PRINT A$
110 NEXT I
120 CLOSE 2
OK
RUN
ENTER THE NAME OF A PRESIDENT? GEORGE WASHINGTON
ENTER THE NAME OF A PRESIDENT? THOMAS JEFFERSON
ENTER THE NAME OF A PRESIDENT? ABRAHAM LINCOLN
GEORGE WASHINGTON
THOMAS JEFFERSON
ABRAHAM LINCOLN
OK
```

Example 3: This example uses the string functions LEFT$, MID$ and RIGHT$ to separate the variable A$ into three string variables. The MKS$ function converts the price of .49 to string format. The CVS function converts the string back to numeric format before increasing the price by ten cents.

```
LIST
10 OPEN "R",1,"PRGM1"
20 FIELD #1, 21 AS A$,4 AS B$,7 AS C$
30 LSET A$="APPLES PEARS  ORANGES"
40 RSET B$=MKS$(.49)
50 LSET C$="A POUND"
60 PUT #1,5
70 GET #1,5
80 X$=LEFT$(A$,7)
90 Y$=MID$(A$,8,7)
100 Z$=RIGHT$(A$,7)
110 P=CVS(B$)+.1
120 PRINT X$,Y$,Z$,P;" CENTS ";C$
130 CLOSE 1
140 END
OK
RUN
APPLES        PEARS        ORANGES        .59  CENTS A POUND
```

LOC Function Form: LOC(file#)

The LOC function returns the record number that will be used if a GET or PUT is executed with the record number parameter omitted. For example:

```
PRINT LOC(1)
Ok
15
```

In the above example, the LOC statement returns the number 15 indicating that the next record to be read or written is record #15.

The LOC function may also be used with Sequential Access files.

LOF Function Form: LOF(file#)

The LOF function returns the number of records read or written in the last extent. An extent is an indexed entry in the ADOS diskette directory. It points to the location of the data on the diskette. An extent holds 128 Random Access blocks or records.

1 Random Access record = 128 bytes = 1 sector

128 sectors = 1 extent

A Random Access file containing more than 128 records requires another extent -- another directory entry.

The LOF function returns a number from 1 to 128. In the example below, 47 records have been read or written in the last extent.

```
PRINT LOF(2)
Ok
47
```

An attempt to use LOF on Sequential Access files will cause a BAD FILE MODE error.

EOF Function Form: EOF(file#)

The EOF function may now be used to detect the end of a Random Access file. If a GET is done past end of file, EOF (file#) will be set equal to -1.

USING ADDS*BASIC™ WITH ADDS*PLUS™ FILES

The following statements are designed to allow ADDS*BASIC™ to operate on files created with ADDS*PLUS™. These statements will provide the ability to read and update up to eight open ADDS*PLUS™ files.

Opening an ADDS*PLUS™ File

ADDS*PLUS™ files are OPENed just like Sequential and Random Access files.

```
OPEN "D",1,"APLUS"
```

After the above OPEN statement, the file is positioned to the first page of the file labeled APLUS.

Using ADDS*PLUS™ files not opened with mode "D" will cause a BAD FILE MODE error as will using a Sequential or Random Access file OPENed with a "D" mode. For "D" mode, the buffer contains the first sector of the current page after OPEN, NEXT and RESET. It contains the last sector of the current page after PUT and GET.

Closing an ADDS*PLUS™ File

ADDS*PLUS™ files must be closed when I/O operations are completed. To CLOSE an ADDS*PLUS™ file, use the CLOSE statement as described previously.

```
CLOSE[file number,[file number...]]
```

ADDS*PLUS™ File I/O

Each page of an ADDS*PLUS™ file is read into a string array. This array must be dimensioned with a DIM statement which determines the number of lines that will be displayed. A page of data can be displayed in any number of lines. However, for legibility and ease in updating, it is advisable to format the page so that it resembles the page displayed under ADDS*PLUS™.

Each element of the array must be given the desired length with the SPACE$ function. The length of each element may vary.

If the ETB character (w) indicating the end of the text is displayed, the array size is greater than the logical size of the form.

The number of characters to be written to disk must be equal to the logical size of the form before an update can take place. To update an ADDS*PLUS™ file the entire page that has been read into an array must be written back to disk. Each element of that array may be modified or changed before the write takes place. However, the new array element (line) must contain exactly the same number of characters as the existing line and the data items must be placed in the appropriate columns, otherwise the data will be out of place when displayed with its form under ADDS*PLUS™.

ADDS+...GET

General Form: ADDS+,[#]file#,GET,string array name,character count

string array name:
one string array variable name

character count:
a constant or multiplication expression containing two constants separated by an asterisk which when evaluated is less than or equal in size to the specified string array. Variables may not be used in the expression.

Purpose: To read a page of an ADDS*PLUS™ disk file into memory.

Remarks: A page of an ADDS*PLUS™ file can be read into an array of any size. If the logical size of the form (the number of characters and blanks up to but not including the ETB character, w) is unknown, use 24*80 or 1920, the maximum size of the screen.

If the character count is larger than the total number of characters in all the elements of the array, a FIELD OVERFLOW error will occur.

Reading stops when the number of characters specified by the character count have been read.

The string array must be dimensioned with a DIM statement and each element of the array must be given the desired length as follows:

```
B$(1)=SPACE$(15)
```

The above statement sets the length of the first element of the string array B$ to 15 characters. The characters in each element are successively replaced by characters read from the disk. If the character count is greater than the logical size of the form, the ETB is stored in the string array.

If the character count = Ø, a FUNCTION CALL error will result.

Examples:

```
ADDS+,#1,GET,B$,16*40

ADDS+,#1,GET,B$,640
```

See EOF function.

ADDS+...NEXT

General Form: ADDS+,[#] file number,NEXT

Purpose: To position the pointer to the next page in the ADDS*PLUS™ file.

Remarks: The ADDS+...NEXT command is used to read through a file. The EOF (file #) should be checked after calls to NEXT. (See EOF function).

Example: See EOF function.

ADDS+...PUT

General Form: ADDS+,[#]file#,PUT,string array name,character count

string array name:
one string array variable name

character count:
a constant or multiplication expression containing two constants separated by an asterisk which when evaluated is less than or equal in size to the specified string array. Variables may not be used in the expression.

Purpose: To update a page of an ADDS*PLUS™ file.

Remarks: To avoid any unnecessary difficulties, the character count should be equal to the logical size of the form. To determine the logical size of the form, read a page into an oversize array and count the number of characters and blanks up to but not including the ETB character (w).

If the current page cannot accommodate the specified number of characters or if the character count is greater than the size of the array, a FIELD OVERFLOW error results.

Since PUT and GET do not change the file position, PUT followed by GET will read the page just written.

Examples: ADDS+,#1,PUT,B$,16*40

ADDS+,#1 PUT,B$,640

ADDS+...RESET

General Form: ADDS+,[#]file #,RESET

Purpose: To position the page pointer to the first page in an ADDS*PLUS™ file.

Remarks: Causes first extent to be reopened if file is positioned on another extent.

Example: See EOF function.

EOF Function Form: EOF(file#)

The EOF function may also be used with ADDS*PLUS™ files to determine if the file is positioned at its end. Recall that EOF returns TRUE (-1) when there is no more data in the file, FALSE (Ø) otherwise.

Example 1: This program reads and prints out all the data in the ADDS*PLUS™ file INVT1. The form used for this file appears below.

```
  ITEM            DESCRIPTION            PRICE

③▨③----③\ⓛ------------------------③\Ⓜ----③▨<
```

Notice that this form contains only one line of data. This line contains 36 characters, 32 characters for the variable fields plus two DUP and two backslash characters used to create the form for sorting. This is the logical size of the form.

Look at the program listing below. Line 40 sets the one element of array B$ to 80 characters and line 80 reads one page of the file into array B$. When this program is executed, the ETB character (w) will appear on the screen. A form can be read into any array of any size. However, the character count must be equal to the logical size of the form before the form can be updated by a PUT statement. When a form is read into an array of the correct size the ETB character will not appear on the screen.

```
LIST
10 CLEAR 400
20 DIM B$(1)
30 OPEN "D",#1,"B:INVT1"
40 B$(1)=SPACE$(80)
50 IF EOF(1) GOTO 100
60 ADDS+,1,GET,B$,80
70 PRINT B$(1)
80 ADDS+,1,NEXT
90 GOTO 50
100 END
OK

RUN
▨1022\SPORTCRAFT BOCCE SET     \7000▨w
▨1145\AUDIOVOX CASSETTE PLAYER\7995▨w
▨1185\TRAVEL ALARM             \ 750▨w
▨1227\ELGIN CALENDAR WATCH     \8995▨w
▨2180\POLAROID SX-70           \2195▨w
^C
BREAK IN 80
OK
```

Example 2: This program updates the same ADDS*PLUS™ file INVT1. A part number and new price is entered and if the part number is valid the price of the item is changed in the ADDS*PLUS™ data file.

```
 10 CLEAR 400
 20 DIM B$(1)
 30 OPEN "D",#1,"B:INVT1"
 40 B$(1)=SPACE$(36)
 50 INPUT "ENTER PART NUMBER AND NEW PRICE";X$,Y$
 60 IF EOF(1) GOTO 260
 70 ADDS+,1,GET,B$,36
 80 IF MID$(B$(1),2,4) = X$ THEN GOTO 110
 90 ADDS+,1,NEXT
100 GOTO 60
110 PRINT
120 PRINT B$(1),"ORIGINAL RECORD"
130 B$(1)=LEFT$(B$(1),31)+Y$+RIGHT$(B$(1),1)
140 PRINT
150 PRINT B$(1),"UPDATE"
160 ADDS+,1,PUT,B$,36
170 ADDS+,1,GET,B$,36
180 ' A PUT FOLLOWED BY A GET
190 ' READS THE PAGE JUST WRITTEN
200 PRINT
210 PRINT B$(1),"NEW RECORD"
220 ADDS+,1,RESET
230 PRINT
240 INPUT "ENTER C TO CONTINUE, T TO TERMINATE";G$
250 IF G$="T" GOTO 280 ELSE GOTO 50
260 PRINT "NO SUCH PART NUMBER"
270 GOTO 220
280 CLOSE
290 END

RUN
ENTER PART NUMBER AND NEW PRICE? 1227,7995

▨ 1227\ELGIN CALENDAR WATCH    \8995 ▨     ORIGINAL RECORD

▨ 1227\ELGIN CALENDAR WATCH    \7995 ▨     UPDATE

▨ 1227\ELGIN CALENDAR WATCH    \7995 ▨     NEW RECORD

ENTER C TO CONTINUE, T TO TERMINATE? T
OK
```

START-UP PROCEDURES

Refer to the ADOS Manual for instructions in bringing up the **Command Processor**. After the prompt (A>) is returned by ADOS, the ADDS*BASIC™ interpreter may be invoked by typing†

A>BASIC or

A>BASIC [file name][/F:# of files][/M:highest memory location]

The file name option indicates that the specified file is to be RUN or executed after initialization is complete. If the file-type is omitted, a default extension of BAS is used. This option allows BASIC programs to be executed in batch mode using the SUBMIT facility of ADOS. Such programs should include a SYSTEM statement which returns control to the **Command Processor** and allows the next program in the batch to execute.

The /F: number of files option sets the number of disk data files that may be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes of memory. If the /F option is omitted, the number of files defaults to 3.

The /M: highest memory location option sets the highest memory location that will be used by BASIC. BASIC requires 21K of memory. If assembly language routines are to be used, the user must reserve the topmost locations in memory. If the /M option is omitted, all of available memory is used.

NOTE: Both the number of files and the highest memory location are numbers that may be either decimal, octal (if preceded by &O), or hexadecimal (preceded by &H).

Examples:

A>BASIC	Use all memory and 3 files, invoke the BASIC Interpreter.
A>BASIC PAYROLL.BAS	Use all memory and 3 files, load and execute PAYROLL.BAS.
A>BASIC INVENT/F:6	Use all memory and 6 files, load and execute INVENT.BAS.

†If ADDS*PLUS™ files will not be accessed, the user may type SBASIC to invoke a smaller Interpreter giving 1K more of user memory space.

A>BASIC /M:32768 Use first 32K of memory and 3 files.

A>BASIC DATACK/F:2/M:&H9000

Use first 36K of memory, 2 files, and execute DATACK.BAS.

The following message appears on the screen when the BASIC Interpreter has been loaded successfully:

```
nnnnn Bytes free
BASIC Rev.n.m
[ADDS*BASIC™ SYSTEM/70]
Copyright 1977 (c) by Microsoft
```

This is immediately followed by the BASIC prompt

```
Ok
```

ENTERING A PROGRAM

The NEW statement

```
NEW
```

should be used before entering a new program, but is not necessary before entering the first program after the BASIC Interpreter has been invoked. This statements deletes the current program and clears all variables (see Section 3).

If the user wishes to have line numbers generated automatically, the AUTO statement may be used (see Section 3).

```
user entry          AUTO
computer response   10 _
```

The user may then enter a BASIC program.

The ENTER key must be depressed after each line is typed to enter the line in memory.

If the AUTO statement is used, every input line is numbered until the CTRL-C keys are depressed.

When entering a program, errors on the current line may be corrected by using the cursor control keys ->, <- and HOME to move the cursor backwards and forwards along the line and the INS CHAR and DEL CHAR keys to modify the line. The ERASE key erases the data from the current cursor position to the end of the logical line.

Enter the program below. This program calculates the roots of a quadratic equation.

```
10 PRINT CHR$(12)
20 PRINT "CALCULATING THE ROOTS OF A QUADRATIC EQUATION"
30 INPUT "ENTER COEFFICIENTS:";A,B,C
40 PRINT " A";"  B";"  C","ROOTS","X1";TAB(47);"X2"
50 IF A=0 THEN 130
60 D=B*B-4*A*C
70 IF D<0 THEN 180
80 IF D=0 THEN 150
90 X1=(-B+SQR(D))/(2*A)
100 X2=(-B-SQR(D))/(2*A)
110 PRINT A;B;C,X1;TAB(47);X2
120 GOTO 30
130 PRINT A;B;C,"UNDEFINED"
140 GOTO 30
150 X1=-B/(2*A)
160 PRINT A;B;C,"EQUAL",X1;TAB(47);X1
170 GOTO 30
180 X1=-B/(2*A)
190 X2=SQR(-D)/(2*A)
200 PRINT A;B;C,"IMAGINARY",X1;"+";X2;"I";TAB(47);X1;"-";X2;"I"
210 GOTO 30
220 END
```

LISTING A PROGRAM

The LIST statement

```
LIST
```

will cause the entire program currently stored in memory to be displayed on the screen. The LIST statement has several options:

```
LIST line number
```

lists the specified line number

```
LIST line number -
```

lists all line numbers greater than or equal to the specified line number.

```
LIST line number - line number
```

lists all line numbers from the first to the second, inclusive.

```
LIST - line number
```

lists all line numbers from the beginning of the program up to and including the specified line number.

EDITING A PROGRAM

Inserting Lines

Additional lines may be inserted by typing the statements with appropriate line numbers. For example, a line inserted between lines 120 and 130 must be given a line number in the range 121 to 129. If this is followed by the LIST statement the user will note that the new line has been placed in the proper sequence in the program. To insert a PRINT statement between lines 30 and 40 of the above program key in

```
35 PRINT      <depress ENTER key>
```

and then LIST the program.

```
LIST
10 PRINT CHR$(12)
20 PRINT "CALCULATING THE ROOTS OF A QUADRATIC EQUATION"
30 INPUT "ENTER COEFFICIENTS:";A,B,C
35 PRINT
40 PRINT " A";"  B";"  C","ROOTS","X1";TAB(47);"X2"
50 IF A=0 THEN 130
60 D=B*B-4*A*C
70 IF D<0 THEN 180
80 IF D=0 THEN 150
90 X1=(-B+SQR(D))/(2*A)
100 X2=(-B-SQR(D))/(2*A)
110 PRINT A;B;C,X1;TAB(47);X2
120 GOTO 30
130 PRINT A;B;C,"UNDEFINED"
140 GOTO 30
150 X1=-B/(2*A)
160 PRINT A;B;C,"EQUAL",X1;TAB(47);X1
170 GOTO 30
180 X1=-B/(2*A)
190 X2=SQR(-D)/(2*A)
200 PRINT A;B;C,"IMAGINARY",X1;"+";X2;"I";TAB(47);X1;"-";X2;"I"
210 GOTO 30
220 END
OK
```

Deleting Lines

Single lines may be deleted by typing each line number followed by the ENTER key. To delete line 220 key in

220 <depress ENTER key>

and then LIST the program.

```
LIST
10 PRINT CHR$(12)
20 PRINT "CALCULATING THE ROOTS OF A QUADRATIC EQUATION"
30 INPUT "ENTER COEFFICIENTS:";A,B,C
35 PRINT
40 PRINT " A";"  B";"  C","ROOTS","X1";TAB(47);"X2"
50 IF A=0 THEN 130
60 D=B*B-4*A*C
70 IF D<0 THEN 180
80 if D=0 THEN 150
90 X1=(-B+SQR(D))/(2*A)
100 X2=(-B-SQR(D))/(2*A)
110 PRINT A;B;C,X1;TAB(47);X2
120 GOTO 30
130 PRINT A;B;C,"UNDEFINED"
140 GOTO 30
150 X1=-B/(2*A)
160 PRINT A;B;C,"EQUAL",X1;TAB(47);X1
170 GOTO 30
180 X1=-B/(2*A)
190 X1=SQR(-D)/(2*A)
200 PRINT A;B;C,"IMAGINARY",X1;"+";X2;"I";TAB(47);X1;"-";X2;"I"
210 GOTO 30
OK
```

To delete more than one BASIC line see the DELETE statement, Section 3.

PRINT FORM Switch

Once a program has been LISTED, any editing causes the scrolling up of the program on the screen. This necessitates numerous reLISTings of the same program. For this reason, it is advantageous to use the single line TTY feature. When the PRINT FORM switch is in the ON position, line 25 behaves like a single line TTY or screen. Scrolling of the previous 24 lines of the display is inhibited and editing of these 24 lines may proceed uninterrupted.

After editing is complete, move the PRINT FORM switch to the OFF position and reLIST the program.

Modifying Lines

The EDIT Statement

The EDIT statement allows modifications and additions to be made to existing program lines without having to retype the entire line each time (see Section 3).

To EDIT a line, type EDIT followed by the line number. The entire line to be EDITed will be printed and the cursor will be positioned under the first character of the statement.

```
EDIT 110
110 PRINT A;B;C,X1;TAB(47);X2
```

The cursor control keys <- -> are used to move backwards or forwards, character by character along the statement. The cursor is never positioned under the line number. The HOME key returns the cursor to the first character in the statement.

Both the FWD TAB and BACK TAB keys move the cursor more quickly along the statement.

To delete a character, depress the DEL CHAR key. To insert one or more characters, depresss the INS CHAR key and key in the characters to be inserted. If editing is complete, depress the ENTER key. To continue editing depress the INS CHAR key again. The INS CHAR key is a lighted toggle action key. To change a character simply key in the new character.

Modify line 110 to read

```
110 PRINT A;B;C,"UNEQUAL",X1;TAB(47);X2
```

Move the PRINT FORM switch to the OFF position before LISTing the program.

The DEL LINE key is used to delete lines in a multiple line statement (see Section 2). When editing a multiple line statement all the cursor control keys function within the line being edited. Depressing the DEL LINE key causes the entire line above the cursor from the (leftmost) position to be deleted.

CTRL-A

The special function keys CTRL-A permit the user to recover a previously typed direct statement. Depressing CTRL-A will cause the system to enter the EDIT mode. The user may then use the INS CHAR key to add a line number to the statement. When the ENTER key is depressed the line is inserted in its proper place in the program (see Edit statement, Section 3).

Depression of these keys may also be used to move a statement by changing the line number. Type EDIT followed by the line number of the statement to be moved. Depress ENTER followed by CTRL-A. The system enters the EDIT mode and the statement minus its line number appears on the screen. The INS CHAR key may then be used to add a new line number to the statements. Remember to delete the original line.

Storing a Program

Once a program has been written, it is often desirable to save it on disk for use at a later time.

```
SAVE "PGM1.BAS"
```

will save the BASIC program PGM1 in compressed binary form on drive A.

```
SAVE "B:PGM1.BAS"
```

will save it in the same format on drive B.

If the program is to be listed under ADOS (see TYPE command) or to be MERGEd with another BASIC program or edited with the ADOS editors it must be saved in ASCII format. This is accomplished by including the A option in the SAVE statement. See Section 9 for a description of valid file names and the SAVE statement.

```
SAVE "PGM1.BAS",A
```

will save the BASIC program in ASCII format.

Loading a Program

Once a program has been saved on disk it may be read into memory with the LOAD statement.

```
LOAD "PGM1.BAS"
```

loads the specified program file into memory from disk drive A and

```
LOAD "B:PGM1.BAS"
```

loads the specified program file into memory from disk drive B (see Section 9, LOAD statement).

Executing a Program

The RUN statement is used to execute a program stored in memory or on disk.

```
RUN
```

(See Run statement, Section 9).

Execute the program you have just entered. Your output should be similar to the output below.

```
CALCULATING THE ROOTS OF A QUADRATIC EQUATION
ENTER COEFFICIENTS: ? 4,7,1

 A  B  C    ROOTS          X1                      X2
 4  7  1    UNEQUAL        -.15693                 -1.59307
ENTER COEFFICIENTS: ? 0,3,7

 A  B  C    ROOTS          X1                      X2
 0  3  7    UNDEFINED
ENTER COEFFICIENTS: ? 4,5,3

 A  B  C    ROOTS          X1                      X2
 4  5  3    IMAGINARY      -.625 + .599479 I       -.625 -.599479 I
ENTER COEFFICIENTS:  ? 8,8,2

 A  B  C    ROOTS          X1                      X2
 8  8  2    EQUAL          -.5                     -.5
ENTER COEFFICIENTS: ? ^C
BREAK IN 30
OK
```

Hard Copy Program Listings - Output

A program listing may be obtained with the LLIST command when the program has been loaded into memory.

```
LLIST[[line number][-line number]]]
```

The options are the same as for the LIST command.

If the program is less than 24 lines, it will fit on the screen and a printed listing can be produced by depressing the PRINT LOCAL key.

Output may also be printed by depressing the PRINT LOCAL key following execution of the program or by specifying the LPRINT statement instead of the PRINT statement (See LPRINT, Section 3).

APPENDIX B :

ASCII CHARACTER CODES

DECIMAL	CHAR.	DECIMAL	CHAR.
000	NUL(CTRL-SHIFT - @)	043	+
001	SOH(CTRL-A)	044	,(comma)
002	STX(CTRL-B)	045	-
003	ETX(CTRL-C)	046	.
004	EOT(CTRL-D)	047	/
005	ENQ(CTRL-E)	048	0
006	ACK(CTRL-F)	049	1
007	BEL(CTRL-G)	050	2
008	BS(CTRL-H)	051	3
009	HT(CTRL-I)(TAB)	052	4
010	LF(CTRL-J)	053	5
011	VT(CTRL-K)	054	6
012	FF(CTRL-L)	055	7
013	CR(CTRL-M)	056	8
014	SO(CTRL-N)	057	9
015	SI(CTRL-O)	058	:
016	DLE(CTRL-P)	059	;
017	DC1(CTRL-Q)	060	<
018	DC2(CTRL-R)	061	=
019	DC3(CTRL-S)	062	>
020	DC4(CTRL-T)	063	?
021	NAK(CTRL-U)	064	@
022	SYN(CTRL-V)	065	A
023	ETB(CTRL-W)	066	B
024	CAN(CTRL-X)	067	C
025	EM(CTRL-Y)	068	D
026	SUB(CTRL-Z)	069	E
027	ESCAPE(CTRL-[)	070	F
028	FS(CTRL-\)	071	G
029	GS(CTRL-])	072	H
030	RS(CTRL-SHIFT-^)	073	I
031	US(CTRL-_)	074	J
032	SPACE	075	K
033	!(FORM SELECT)	076	L
034	"	077	M
035	#	078	N
036	$	079	O
037	%	080	P
038	&	081	Q
039	'(single quote)	082	R
040	(	083	S
041	)	084	T
042	*	085	U

LF=NEW LINE FF=FORM FEED CR=ENTER DEL=ERASE

DECIMAL	CHAR.
086	V
087	W
088	X
089	Y
090	Z
091	[
092	\
093	]
094	^
095	_
096	`
097	a
098	b
099	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~

DECIMAL	CHAR.
127	DEL(ERASE)
130	↑
132	←
134	→
136	↓
138	HOME
140	SHIFT-ERASE
142	PRINT LOCAL
144	INS CHAR
146	DEL CHAR
148	FWD TAB
150	BACK TAB
154	INS LINE
159	DEL LINE
160	REVERSE VIDEO
164	FULL INTENSITY
168	HALF INTENSITY
172	UNDERLINE
176	REVERSE VIDEO BLINK
180	FULL INTENSITY BLINK
184	HALF INTENSITY BLINK
188	ZERO INTENSITY

The following table of statements is listed in alphabetical order. The statements which direct ADDS*BASIC™ to arrange memory and input/output facilities, to list and edit programs, and to handle other housekeeping details in support of program execution are usually entered only in direct mode. However, all statements with the exception of DEF FN and DEFUSR may also be used in direct mode.

The format for an ADDS*BASIC™ line is as follows:

```
nnnnn statement [:statement...]
```

where nnnnn is the line number.

In the table below, the following abbreviations are used:

drv:	denotes the disk drive to be used; if omitted, drive A is assumed
gfn:	denotes a general file name
sfn:	denotes a specific file name
nnnnn:	indicates a line number
N:	indicates a file number (1-8)
M:	indicates field size (1-128)
L:	indicates record number (1-1900)
[]:	indicates optional items
X,Y:	denote expressions
I,J:	denote expressions whose values are truncated to integers
V,W:	denote variable names

Table C.1. Summary of Statements

Statement	Format	Description
ADDS+,...,GET	ADDS+,[#]N,GET,V$,character count	Reads the current page of the ADDS*PLUS™ data file into array V$ in memory. Reading stops when the number of characters specified by the character count have been read. The character count may be a numeric expression (row*column) or a numeric constant. Variables must not be used to denote the character count. The size of each element of V$ must be declared by using the string function SPACE$. V$(I)=SPACE$(K) Note: A page may be read into an array of any size. If the form size is unknown, use 24*80 or 1920, the maximum size of the screen.
ADDS+,...NEXT	ADDS+,[#]N,NEXT	Positions the page pointer to the next page in the ADDS*PLUS™ file. The EOF(file #) should be checked after calls to NEXT.
ADDS+,...PUT	ADDS+,[#]N,PUT,V$,character count	Updates the current page of the ADDS* PLUS™ date file. To avoid any unnecessary difficulties, the character count must be equal to the logical size of the form (the number of data characters and blanks contained in each page of the data file). The character count may be a numeric expression (row*column) or a numeric constant. Variables must not be used to denote the character count.
ADDS+,...RESET	ADDS+,[#]N,RESET	Positions the page pointer to the first page in the ADDS*PLUS™ file.

Table C.1. Summary of Statements (continued)

Statement	Format	Description
AUTO	AUTO [[initial line#][,[increment]]]	Automatically generates line numbers. The following options are available: 1. AUTO - initial line # and increment both default to 10 2. AUTO nnnnn - increment defaults to 10 3. AUTO nnnnn, - increment used in the last AUTO statement is assumed 4. AUTO,I - initial line defaults to Ø
CLEAR	CLEAR [I]	Reserves bytes of string space equal to the value of the expression I and clears program variables. If I is omitted, all program variables are set to zero. When ADDS*BASIC™ is loaded, string space is set to 100 bytes.
CLOSE	CLOSE [Nl[,Nl,...]]	Closes the specified files. If the file numbers are omitted, all files are closed.
CONT	CONT	Continues program execution after a CTRL-C has been depressed, an error has occurred or a STOP or END statement has been executed. Execution resumes at the statement following the break unless input from the terminal was interrupted. In that case, execution resumes with the reprinting of the prompt (?) or the prompt string followed by the prompt (?). CONT is useful in debugging, especially where an infinite loop is suspected. An infinite loop is a series of statements from which there is no escape. Typing CTRL-C causes a break in execution and puts the system at command level. Direct mode statements can then be used to print intermediate values, change the values of variables, etc. Execution can be restarted by typing the CONT command, or by executing a direct mode GOTO statement, which causes execution to resume at the specified line number.

Table C.1. Summary of Statements (continued)

Statement	Format	Description
DATA	DATA list	Specifies the data to be read by a READ statement. List elements can be numbers or strings and must be separated by commas.
DEF	DEF FNT[(V[,W,...])]=X	Defines a user-defined function. Function name is FN followed by a legal variable name. User-defined string functions are allowed. Definitions are restricted to one line (255 characters).
DEFDBL	DEFDBL range of letters	Defines double precision default variable names.
DEFINT	DEFINT range of letters	Defines integer default variable names.
DEFSNG	DEFSNG range of letters	Defines single precision default variable names.
DEFSTR	DEFSTR range of letters	Defines string default variable names.
DEFUSR	DEFUSR (digit) = X	Defines the starting address of assembly language subroutines. Up to ten subroutines are allowed (0-9).

Table C.1. Summary of Statements (continued)

Statement	Format	Description
DELETE	DELETE [[starting line #][-ending line #]]	Deletes lines of the current program. Several options are permitted. 1. If the starting line # is omitted DELETE - nnnnn deletes every line of the current program up to and including the ending line #. 2. If the ending line # is omitted DELETE nnnnn deletes only the specified line. 3. If both line numbers are included, DELETE nnnnn-nnnnn deletes all lines of the current program from the first to the second line # inclusive. An ILLEGAL FUNCTION CALL error occurs if the specified line numbers do not exist.
DIM	DIM V(I[,J...])[,...]	Allocates space for array variables. More than one variable may be dimensioned by one DIM statement up to the limit of the line. The value of each expression gives the maximum subscript possible. The smallest subscript is 0. Without a DIM statement, an array is assumed to have a maximum subscript of 10 for each dimension referenced. For example, A(I,J) is assumed to have 121 elements, from A(0,0) to A(10,10) unless otherwise dimensioned in a DIM statement.
EDIT	EDIT nnnnn	Allows editing of the specified line without affecting any other lines.
END	END	Terminates the execution of a program and closes all files. Control is returned to the command level of BASIC.

Table C.1. Summary of Statements (continued)

Statement	Format	Description
ERASE	ERASE V[,W...]	Eliminates the arrays specified. The arrays may be redimensioned or the space made available for other uses.
ERROR	ERROR I	Forces error with code specified by the expression. Used primarily for user-defined error codes.
FIELD	FIELD #N,M AS V$ [,...]	Associates some or all of a file's random buffer with particular string variables. May be used only with random files, otherwise a BAD FILE MODE error will occur.
FILES	FILES ["[drv:]gfn"]	Lists the names of all files satisfying the given file reference; either a general or specific file name may be used. The command is interrupted by CTRL-C. If the filetype is omitted it is assumed to be ØØØ.
FOR	FOR V = X TO Y [STEP Z]	Allows the repeated execution of the same statements. First execution sets V=X. Execution proceeds normally until NEXT is encountered. Z is added to V, then, IF Z<Ø and V>=Y, or If Z>Ø and V<=Y, control is returned to the statement after FOR. Otherwise, execution continues with the statement after NEXT.

Table C.1. Summary of Statements (continued)

Statement	Format	Description
GET	GET [#]N[,L]	Reads the data from disk to the random buffer. If an attempt is made to GET a record which has never been PUT, all zeros are read into the record and no error occurs. If the record number "L" is omitted, the record number that is one higher than the previous GET is used. Initially, a GET without a record number will read the first record.
GOSUB	GOSUB nnnnn	Unconditional branch to subroutine beginning at line nnnnn.
GOTO	GOTO nnnnn	Unconditional branch to the line number nnnnn.
IF...GOTO	IF X [THEN] GOTO nnnnn	Same as IF...THEN except GOTO can only be followed by a line number and not another statement.
IF...THEN (ELSE)	IF X THEN X [ELSE Y] or IF X THEN statement [:statement...] [ELSE statement [:statement...]]	If value of X<>0 (TRUE), branches to the line number or statement(s) after THEN. Otherwise, branches to the line number or statement(s) after ELSE. IF ELSE is omitted, and the value of X=0 (FALSE), execution proceeds at the line after the IF...THEN.
INKEY	INKEY	Reads a single key value into a variable. A numeric variable will be assigned the decimal which is the ASCII equivalent of the key value. A string variable will be assigned the ASCII character.

Table C.1. Summary of Statements (continued)

Statement	Format	Description
INPUT	INPUT ["prompt string";]V[,W...]	Causes BASIC to request input from the screen during program execution. String or numeric values typed on the terminal are assigned to the variables in the list.
INPUT #	INPUT #N,V[,W...]	Inputs the data from a sequential access disk file. Leading blanks and ENTER key characters are ignored.
KILL	KILL "[drv:]sfn"	Deletes a file from disk and frees the disk space used by the file. If the filetype is omitted it is assumed to be ØØØ.
LET	[LET] V = X	Assigns the value of the expression to the variable. The word LET is optional.
LINE INPUT	LINE INPUT ["prompt string";]V$	Inputs a whole line to a string variable from the terminal during program execution without using quotes, commas or other characters as delimiters.
LINE INPUT #	LINE INPUT #N,V$	Inputs an entire line of a file into a string without using quotes, commas or other characters as delimiters.

Table C.1. Summary of Statements (continued)

Statement	Format	Description
LIST	LIST[[starting line #][-[ending line #]]]	Lists the program currently in memory. Allows several listing options: 1. If both line numbers are omitted, LIST lists the program currently in memory starting with the lowest numbered line. 2. If the starting line # is omitted, LIST -nnnnn lists all lines from the beginning of the program to the specified ending line #, inclusive. 3. If the ending line # is omitted, LIST nnnnn prints the single line if it exists and LIST nnnnn- lists all lines with numbers greater than or = to the specified starting line #. 4. If both numbers are used, LIST nnnnn-nnnnn lists all lines from the starting line number to the ending line #, inclusive. To terminate the LIST command, depress CTRL-C. To cause the listing to pause, depress CTRL-S. Depressing any key except CTRL-C causes the listing to resume.
LLIST	LLIST[[starting line #][-[ending line #]]]	Same as LLIST with the same options. Program is listed on the line printer.
LOAD	LOAD "[drv:]sfn"[,R]	Reads a file from disk and loads it into memory. All variable and program lines in memory are deleted. If the filetype is omitted it is assumed to be BAS. If the file does not exist, a FILE NOT FOUND error will occur. If the "R" or RUN option is used, all data files are kept OPEN. Without the "R" option, all files are automatically closed.
LOCATE	LOCATE I,J	Positions the cursor at the specified row (I) and column (J).

Table C.1. Summary of Statements (continued)

Statement	Format	Description
LPRINT	LPRINT list	Same as PRINT, but prints on the line printer.
LPRINT USING	LPRINT USING "format string"; list	Same as PRINT USING, but prints on the line printer. For a detailed description, see Section 6.
LSET	LSET V$=X$	Assigns a value (left-justified) to a field of specified length in the random buffer.
MERGE	MERGE "[drv:]sfn"	Merges the specified filename stored on disk in ASCII format into the program already in memory. If the file is not in ASCII format, a BAD FILE MODE error will occur. If there are lines in the program on disk which have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding program lines in memory. If the filetype is omitted it is assumed to be BAS.
NAME	NAME "[drv:]oldsfn" AS "[drv:]newsfn"	Changes the name of a file. A FILE NOT FOUND error results if the old filename does not exist, and a FILE ALREADY EXISTS error occurs if a file exists with the new filename. If the filetype is omitted, it is assumed to be ØØØ.
NEXT	NEXT [V,W,...]	Last statement of a FOR loop. V is the variable of the most recent loop, W of the next most recent and so on. NEXT without a variable terminates the most recent FOR loop.

Table C.1. Summary of Statements (continued)

Statement	Format	Description
NEW	NEW	Deletes the current program and clears all variables. Use before entering a new program.
ON ERROR GOTO	ON ERROR GOTO nnnnn	When an error occurs, branches to line specified. Sets the variable ERR to error code and ERL to line number where the error occurred. See Appendix H for a list of error codes. ON ERROR GOTO Ø disables error trapping.
ON...GOSUB	ON I GOSUB list of line numbers	Same as ON...GOTO except list elements are initial line numbers of subroutines.
ON...GOTO	ON I GOTO list of line numbers	Branches to line whose number is Ith in the list. List elements are separated by commas. If I=Ø or > number of elements in the list, execution continues at the next statement. If I<Ø or >255, an ILLEGAL FUNCTION CALL error results.
OPEN	OPEN "mode",[#]N,"[drv:]sfn"	Opens a sequential, random or ADDS*PLUS™ file. If the filetype is omitted it is assumed to be ØØØ. mode O = sequential output mode mode I = sequential input mode mode R = random input/output mode mode D = ADDS*PLUS™ input/output mode The mode may also be a string expression whose first character is one of the above.

Table C.1. Summary of Statements (continued)

Statement	Format	Description
POKE	POKE I,J	Enters data into memory in binary form. Stores byte J in memory location derived from I. 0<=J<=255; 0<I<65536.
PRINT	PRINT list	Causes values of expressions in the list to be printed on the screen. If no list items are specified, a blank line is printed. Spacing is determined by punctuation. A comma either between list items or at the end of the line causes printing to begin at the next 14-column zone. A semi-colon in either position causes printing to begin immediately. A blank between list items acts in the same way as a semi-colon. A blank or any other punctuation at the end of the statement causes printing to begin at the beginning of the next line. String literals may be printed if enclosed by quotation marks.
PRINT USING	PRINT USING "format string";list	Prints the values of the expressions in the list edited according to the format string. The string is an expression which represents the line to be printed. The list contains the constants, variable names or expressions to be printed. List entries are separated by commas or semicolons. See Section 6.
PRINT #	PRINT #N, list	Writes data to a sequential file. All data must be delimited by one of the following characters: ENTER key comma

Table C.1. Summary of Statements (continued)

Statement	Format	Description
PRINT # USING	PRINT #N, USING "format string"; list	Writes the data to a sequential file using the format specified.
PUT	PUT [#]N[,L]	Writes the data in the random buffer to disk. If the record number is omitted, the data is written to the record number that is one higher than the previous PUT statement. Initially, a PUT without a record number "L" will write the first record.
READ	READ V[,W...]	Assigns values in DATA statements to variables. Values are assigned in sequence starting with the first value in the first DATA statement.
REM	REM [remarks]	Allows the insertion of comments. Not executed, but may be branched into. Remarks may also be added to the end of a line preceded by a single quotation mark ('').
RENUM	RENUM [[new line #],[present line #], [increment]]	Renumbers an entire program or a segment of a program stored in memory. The new line #, present line # and increment are all optional. If the new line # is omitted, it is assumed to be 10. If the present line # is omitted, the entire program will be resequenced. If the increment is omitted it is also assumed to be 10.
RESET	RESET	This statement must be entered immediately after switching disks in any drive. RESET constructs a new map of the directory.

Table C.1. Summary of Statements (continued)

Statement	Format	Description
RESTORE	RESTORE [nnnnn]	Allows data from DATA statement to be reread. Next READ statement after RESTORE begins with first datum of first data statement with specified line number.
RESUME	RESUME [nnnnn]	Resumes program execution at the line specified after error trapping routine. If number is omitted or is zero, resumes at statement where the error occurred. RESUME NEXT causes resumption at the statement following the statement where the error was made.
RETURN	RETURN	Terminates a subroutine. Branches to the statement after the most recent GOSUB.
RSET	RSET V$=X$	Assigns a value, right-justified, to a string variable or a field of specified length in the random buffer.
RUN	RUN [nnnnn]	Starts execution of the program currently in memory at the line specified. If the line number is omitted, execution begins at the lowest line number.
RUN	RUN "[drv:]sfn"[,R]	Starts execution of the specified file name on the specified disk drive. If the disk drive is omitted, it is assumed to be A. All files are closed unless ",R" is specified after the file name. If filetype is omitted, it is assumed to be BAS.

Table C.1. Summary of Statements (continued)

Statement	Format	Description
SAVE	SAVE "[drv:]sfn"[,A]	Saves the program in memory on disk. If the A option is present, the file is saved in ASCII format; otherwise, it is saved in a compressed binary format. If the filetype is omitted, it is assumed to be BAS.
STOP	STOP	Stops program execution. BASIC enters command level and prints BREAK IN LINE nnnnn. Unlike END, STOP does not close files.
SWAP	SWAP V,W	Exchanges values of the variables named. Variables must be of the same type.
SYSTEM	SYSTEM	To be used with BASIC programs which are to be executed in batch mode. This statement closes all files and returns control to the **Command Processor** to allow the next program to be executed.
TROFF	TROFF	Turns off trace flag. The trace flag is turned on by TRON (see below). NEW also turns off the trace flag.
TRON	TRON	Turns on trace flag. Prints number of each line in square brackets as it is executed.
WAIT	WAIT I	System waits or pauses for 1/10 second times the specified integer constant before continuing.

ADDS*BASIC™ provides several commonly used algebraic and string functions which may be called from any program without further definition. The functions in the following table are listed in alphabetical order. As usual, X and Y stand for expressions, I and J for integer expressions and X$ and Y$ for string expressions.

Table D.1. Summary of Intrinsic Functions

Function	Call Format	Description
ABS	ABS(X)	Returns absolute value of expression X. ABS(X)=X if X>0, -X if X<0.
ASC	ASC(X$)	Returns the ASCII code of the first character of the string X$. ASCII codes are in Appendix B.
ATN	ATN(X)	Returns arctangent (X). Result is in radians in range -PI/2 to PI/2. (PI/2=1.5708)
CDBL	CDBL(X)	Converts X to double precision.
CINT	CINT(X)	Converts X to integer. If the argument is in the range -32768 to 32767, the CINT(X)=INT(X). Otherwise, CINT will produce an OVERFLOW error.
COS	COS(X)	Returns COS(X). X is in radians.
CSNG	CSNG(X)	Converts X to single precision.
CHR$	CHR$(I)	Returns a string whose one element has ASCII code I. ASCII codes are in Appendix B. 0<=I<=255
CVD	CVD (eight byte string)	Converts a packed string representation of a number input from a random access file to a double precision value.
CVI	CVI (two byte string)	Converts a packed string representation of a number input from a random access file to an integer value.
CVS	CVS (four byte string)	Converts a packed string representation of a number input from a random access file to a single precision value.

Table D.1. Summary of Intrinsic Functions (continued)

Function	Call Format	Description
EOF	EOF(file #)	End of file detection for sequential, random and ADDS*PLUS™ files. Returns TRUE (-1) if there is no more data in the file and FALSE if more data is available. If an attempt is made to input past the end of a data file, an INPUT PAST END error will occur.
ERL	ERL	Returns the number of the line in which the last error occurred.
ERR	ERR	Returns the error code of the last error.
EXP	EXP(X)	Returns e to the power X where e=2.71828 and X must be <=87.3365.
FIX	FIX(X)	Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.
FRE	FRE(0)	Returns number of bytes in memory not being used by BASIC.
	FRE(X$)	Returns the number of bytes of remaining free string space.
HEX$	HEX$(X)	Returns a string which represents the hexadecimal of the decimal argument.
INPUT$	INPUT$(X[,[#]file#])	Returns a string of X characters read in from a sequential access file. If X is too large, an INPUT PAST END error will occur. If the file number is omitted, INPUT$(X) returns a string of X characters input from the screen during program execution. No prompt is printed and characters entered are not echoed back to the screen. The INPUT$ function is terminated by CTRL-C.

Table D.1. Summary of Intrinsic Functions (continued)

Function	Call Format	Description
INSTR	INSTR([I,]X$,Y$)	Searches for the first occurrence of string Y$ in X$ and returns the position. Optional offset I sets position for starting the search. 1<=I<=255. If I>LEN(X$), if X$ is null or if Y$ can not be found, INSTR returns Ø. If Y$ is null INSTR returns I unless I>LEN(X$) in which case Ø is returned. Strings may be string variable values, string expressions or string literals.
INT	INT(X)	Returns the largest integer <=X. For example, INT(.23)=Ø, INT(-.23)=-1. The following would round X to D decimal places: INT(X*10 D+.5)/10 D.
LEFT$	LEFT$(X$,I)	Returns leftmost I characters of string X$. 0<=I<=255.
LEN	LEN(X$)	Returns length of string X$. Non-printing characters and blanks are counted as part of the length.
LOC	LOC(file #)	Determines the current record number of a random file or the sector number of a sequential file.
LOF	LOF(file #)	Determines the number of records read or written in the last extent (1-128) for random files only.
LOG	LOG(X)	Returns natural log of X. X>0.
LPOS	LPOS(I)	Returns the current position of the line printer print head within the line printer buffer. The expression I must be given, but the value is ignored.
MID$	MID$(X$,I[,J])	Without J, returns rightmost characters from X$ beginning with the Ith character. If I>LEN(X$), MID$ returns the null string. 0<=I<=255. With 3 arguments, returns a string of length J of characters from X$ beginning with the Ith character. If J is greater than the number of characters in X$ to the right of I, MID$ returns the rest of the string. 0<=J<=255.

Table D.1. Summary of Intrinsic Functions (continued)

Function	Call Format	Description
MKD$	MKD$ (double precision value)	Converts a number to the packed representation of the number as an eight byte string for storage in a random access file.
MKI$	MKI$ (integer value)	Converts a number to the packed representation of the number as a two byte string for storage in a random access file. The fractional part of the value is lost. An OVERFLOW ERROR occurs if the value is outside integer range.
MKS$	MKS$ (single precision value)	Converts a number to the packed representation of the number as a four byte string for storage in a random access file.
OCT$	OCT$(X)	Returns a string which represents the octal value of the decimal argument.
PEEK	PEEK(I)	Retrieves data from memory location I in binary form. I is chosen in the same way as in the POKE statement. The value returned is an integer between 0 and 255. PEEK is used to return arguments and results from assembly language programs.
POS	POS(I)	Returns present column position of terminal's print head, where the leftmost position=1. As with LPOS, I is ignored.
RIGHT$	RIGHT$(X$,I)	Returns rightmost I character of string X$. If I=LEN(X$) returns X$. 0<=I<=255.
RND	RND(X)	Returns a random number between Ø and 1. X<Ø starts a new sequence of random numbers. X>Ø gives the next random number in sequence. X=Ø gives the last number returned. Sequences started with the same negative number will be the same.

Table D.1. Summary of Intrinsic Functions (continued)

Function	Call Format	Description
SGN	SGN(X)	If X>Ø, returns 1, if X=Ø returns Ø, if X<Ø, returns -1. For example, ON SGN(X)+2 GOTO 100,200,300 branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.
SIN	SIN(X)	Returns the sine of the value of X in radians. Note that 1 Radian = 180/PI degrees = 57.2958 degrees, so that the sin of X degrees = sin (X/57.2958).
SPACE$	SPACE$(I)	Returns a string of spaces of length I. 0<=I<=255.
SPC	SPC(I)	Prints I blanks on terminal. 0<=I<=255. May only be used in PRINT and LPRINT statements.
SQR	SQR(X)	Returns square root of X. X must be >=0.
STR$	STR$(X)	Returns string representation of numeric value of X.
STRING$	STRING$(I,J)	Returns a string of length I whose characters all have ASCII code J. See Appendix B for ASCII codes. 0<=I<=255.
	STRING$(I,X$)	Returns a string of length I containing the first character of X$.
TAB	TAB(I)	Spaces to position I on the terminal. Space 1 is the leftmost space, 80 the rightmost. If the carriage is already beyond space I, TAB has no effect. 1<=I<=255. May only be used in PRINT and LPRINT statements.
TAN	TAN(X)	Returns tangent(X). X is in radians.
USR	USRn(X)	Calls the user's assembly language subroutine with argument X. Any of 10 different routines may be called by setting n to Ø,1,...9. The starting address of the user's routine must have been defined in a DEFUSRn statement.

Table D.1. Summary of Intrinsic Functions (continued)

Function	Call Format	Description
VAL	VAL(X$)	Returns numerical value of string X$. If first character of X$ is not +, -, a digit or decimal point, VAL(X$)=0.
VARPTR	VARPTR(V)	Returns the address of the variable given as the argument. If the variable has not been assigned a value during the execution of the program, an ILLEGAL FUNCTION CALL error will occur. The main use of the VARPTR function is to obtain the address of a variable or array so it may be passed to an assembly language subroutine. Arrays are usually passed by specifying VARPTR(A(0)) so that the lowest addressed element of the array is returned.
VARPTR	VARPTR (file #)	Returns the address of the 128 byte data buffer for the random file # specified.
		NOTE All simple variables should be assigned values in a program before calling VARPTR for any array. Otherwise, allocation of a new simple variable will cause the addresses of all arrays to change.

The following functions, while not intrinsic to ADDS*BASIC™, can be calculated using the existing BASIC functions:

FUNCTION	BASIC EQUIVALENT
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE	ARCSIN(X) = ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X) = -ATN X(X/SQR(-X*X+1)) +1.57Ø8
INVERSE SECANT	ARCSEC(X) = ATN(XSQR(X*X-1)) +SGN(SGN(X)-1)*1.57Ø8
INVERSE COSECANT	ARCCSC(X) = ATN(1/SQR(X*X-1)) +(SGN(X)-1)*1.57Ø8
INVERSE COTANGENT	ARCCOT(X) = ATN(X)+1.57Ø8
HYPERBOLIC SINE	SINH(X) = (EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X) = (EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X) = EXP(-X/EXP(X)+EXP(-X)) *2+1
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X) = EXP(-X)/(EXP(X) - EXP(-X)) *2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X) = LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X) = LOG(X+SQR(X*X+-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X) = LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X) = LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X) = LOG((SGN(X)* SQR(X*X+1)+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X) = LOG((X+1)/(X-1))/2

Some words are reserved by the ADDS*BASIC Interpreter for use as statements or operators, etc. and thus may not be used for variable or function names. The reserved words are listed below:

RESERVED WORDS

AND
AUTO
CLEAR
CLOSE
CONSOLE
CONT
DATA
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
EDIT
ELSE
END
ERASE
ERL
ERR
FIELD
FILES
FN
FOR
GET
GOSUB
GOTO
IF
IMP
INPUT
INSTR
KILL
KL
LET
LINE
LIST
LLIST
LOAD
LPRINT
LSET
MERGE
MOD
MOUNT
NAME
NEW
NEXT
NOT
NULL
ON
OPEN
OR
OUT
PEEK
POKE
PRINT
PUT
READ
REM
RENUM
RESUME
RETURN
RSET
RUN
SPACE$
SPC
STOP
STRING$
SWAP
TAB
THEN
TO
TROFF
TRON
UNLOAD
USR
VARPTR
WAIT
WIDTH
XOR

When the BASIC interpreter detects an error that will cause the program to be terminated, it prints an error message and then returns to the command level and prints Ok. The format of the error messages is as follows:

Direct Mode	Error
Indirect Mode	Error in nnnnn

where:

error is the error message
nnnnn is the line number where the error occurred.

Each statement has its own particular possible errors in addition to the general errors in syntax. Table G.1 gives all the error messages and their meanings.

Table G.1. Error Messages

Error Message	ERR=	Description
SYNTAX ERROR	0	Missing parenthesis in an expression, illegal character in a line, incorrect punctuation, etc.
NEXT WITHOUT FOR	1	The variable in a NEXT statement does not correspond to a previously executed FOR statement.
RETURN WITHOUT GOSUB	3	A RETURN statement was encountered before a previous GOSUB statement was executed.
OUT OF DATA	4	A READ statement was executed but all of the DATA statements in the program have already been read. The program tried to read too much data or insufficient data was included in the program.
ILLEGAL FUNCTION CALL	5	The parameter passed to a math or string function was out of range. This error can occur due to: 1. a negative array subscript (LET A(-1)=0) 2. an unreasonably large array subscript (>32767) 3. LOG with negative or zero argument 4. SQR with negative argument 5. A B with A negative and B not an integer 6. A call to USR before the address of a machine language subroutine has been entered. 7. Calls to MID$, LEFT$, RIGHT$, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR or ON...GOTO with an improper argument.
OVERFLOW	6	The result of a calculation was too large to be represented in ADDS*BASIC™'s number format. If an underflow occurs, zero is given as the result and execution continues without any error message being printed.

Table G.1. Error Messages (continued)

Error Message	ERR=	Description
OUT OF MEMORY	7	Program is too large, has too many variables, too many FOR loops, too many GOSUBs or too complicated expressions. See Appendix I, Space and Speed Hints.
UNDEFINED LINE	8	The line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE was to a line which does not exist.
SUBSCRIPT OUT OF RANGE	9	An attempt was made to reference an array element which is outside the dimensions of the array. This error can occur if the wrong number of dimensions are used in an array reference. For example: LET A(1,1,1)=Z when A has already been dimensioned by DIM A(10,10)
REDIMENSIONED ARRAY	10	After an array was dimensioned, another dimension statement for the same array was encountered. This error often occurs if an array has been given the default dimension of 10 and later in the program a DIM statement is found for the same array.
DIVISION BY ZERO	11	Can occur with integer division and MOD as well as floating point division. Ø to a negative power also causes a DIVISION BY ZERO error.
ILLEGAL DIRECT	12	DEF FN and DEF USR are illegal in the direct mode.
TYPE MISMATCH	13	The left-hand side of an assignment statement was a numeric variable and the right-hand side was a string, or vice-versa; or a function which expected a string argument was given a numeric one or vice-versa.

Table G.1. Error Messages (continued)

Error Message	ERR=	Description
OUT OF STRING SPACE	14	String variables exceed amount of string space allocated for them. Use the CLEAR statement to allocate more string space or use smaller strings or fewer string variables.
STRING TOO LONG	15	An attempt was made to create a string more than 255 characters long.
STRING FORMULA TOO COMPLEX	16	A string expression was too long or too complex. Break it into two or more shorter ones.
CAN'T CONTINUE	17	Attempt to continue a program when none exists or after a modification was made to the program.
UNDEFINED USER FUNCTION	18	Reference was made to a user defined function which had never been defined.
UNPRINTABLE ERROR	19	An error condition exists for which there is no error message available. Probably there is an ERROR statement with an undefined error code.
RESUME WITHOUT ERROR	20	A RESUME statement was encountered, but no error trapping routine had been entered.
NO RESUME	21	BASIC entered an error trapping routine, but the program ended before a RESUME statement was encountered.
MISSING OPERAND	22	During evaluation of an expression, an operator was found with no operand following it.
LINE BUFFER OVERFLOW	23	An attempt was made to input a program or data line which has too many characters to be held in the line buffer. Shorten the line or divide it into two or more parts.

Table G.1. Error Messages (continued)

Error Message	ERR=	Description
POSITION NOT ON SCREEN	24	An attempt was made to move the cursor to a non-existent position on the screen with the LOCATE statement. Either the specified row was outside the range 1 to 24 or the specified column was outside the range 1-80.

Table G.2. Disk Error Messages

Error Message	ERR=	Description
FIELD OVERFLOW	50	An attempt was made to allocate more than 128 characters of string variables in a single FIELD statement.
INTERNAL ERROR	51	Internal error in BASIC. Report conditions under which error occurred and all relevant data to ADDS software department. This error can also be caused by certain kinds of disk I/O errors.
BAD FILE NUMBER	52	An attempt was made to use a file number which specifies a file that is not OPEN or that is greater than the number of files entered during the ADDS*BASIC™ initialization dialog.
FILE NOT FOUND	53	Reference was made in a LOAD, KILL or OPEN statement to a file which did not exist on the disk specified.
BAD FILE MODE	54	An attempt was made to perform a PRINT to a random file, to OPEN a random file for sequential output, to perform a PUT or GET on a sequential file, to load a random file or to execute an OPEN statement where the file mode is not I, O, R, or D.
FILE ALREADY OPEN	55	A sequential output mode OPEN for a file was issued for a file that was already OPEN and had never been CLOSED.
DISK I/O ERROR	57	An I/O error occurred on disk X. A sector read (checksum) error occurred eighteen (18) consecutive times.

Table G.2. Disk Error Messages (continued)

Error Message	ERR=	Description
FILE ALREADY EXISTS	58	The new file name specified in a NAME statement had the same name as another file that already existed on the disk. Try a different name.
SET TO NON-DISK STRING	59	An LSET or RSET was given for a string variable which had not previously been mentioned in a FIELD statement.
DISK FULL	61	All disk storage is exhausted on the disk. Delete some old disk files and try again.
INPUT PAST END	62	An INPUT statement was executed after all the data in a file had been INPUT. This will happen immediately if an INPUT is executed for a null (empty) file. Use of the EOF function to detect End of File will avoid this error.
BAD RECORD NUMBER	63	In a PUT or GET statement, the record number is either greater than the allowable maximum of approximately 1900 or equal to zero.
BAD FILE NAME	64	A file name of Ø characters (null) or a file name whose first byte was Ø or 377 octal (255 decimal) or a file name with more than 8 characters was used as an argument to LOAD, SAVE, KILL or OPEN.
MODE-MISMATCH	65	Sequential OPEN for output was executed for a file that already existed on the disk as a random (R) mode file, or vice versa.
DIRECT STATEMENT IN FILE	66	A direct statement was encountered during a LOAD of a program in ASCII format. The LOAD is terminated.
TOO MANY FILES	67	A SAVE or OPEN (O or R) was executed which would create a new file on the disk, but all 255 directory entries were already full. Delete some files and try again.

Space Allocation

The memory space required for a program depends, of course, on the number and kind of elements in the program. ADDS*BASIC™ itself takes about 21K. Table H.1 contains information on the space required for the various program elements.

Table H.1. Space Allocation

Element	Type	Space Required
Variables	Integer	5 bytes
	Single Precision	7 bytes
	Double Precision	11 bytes
	String	6 bytes
Arrays		(# of elements)*X+6+ (# of dimensions)*2 bytes
		X
	Integer	2
	Single Precision	4
	Double Precision	8
	String	3 + # of characters
Functions	Intrinsic	2 bytes (call)
	User-Defined	6 bytes (definition)
Reserved Words		1 byte each 2 bytes for ELSE
Other Characters		1 byte each
Stack Space	Active FOR loop	17 bytes
	Active GOSUB	5 bytes
	Parentheses	6 bytes (each set)
	Temporary Result	12 bytes

Space Hints

The space required to run a program may be significantly reduced without affecting execution by following a few of the following hints.

1. Use multiple statements per line. Each line has a 5 byte overhead for the line number, etc., so the fewer lines there are, the less storage is required.

2. Delete unnecessary spaces. Instead of writing

   ```
   1Ø PRINT X, Y, Z
   ```

   use

   ```
   1Ø PRINTX,Y,Z
   ```

3. Delete REM statements to save 1 byte for REM and 1 byte for each character of the comment.

4. Use variables instead of constants, especially when the same value is used several times. For example, using the constant 3.14159 ten times in a program uses 4Ø bytes more space than assigning

   ```
   1Ø P=3.14159
   ```

   once and using P ten times.

5. Using END as the latest statement of a program is not necessary and takes one extra byte.

6. Reuse unneeded variables instead of defining new variables.

7. Use subroutines instead of writing the same code several times.

8. Use the zero elements of arrays. Remember, the array dimensioned by

   ```
   100 DIM A(10)
   ```

   has eleven elements, A(0) through A(10).

9. Use integer variables whenever possible.

Speed Hints

1. Deleting spaces and REM statements gives a small but significant decrease in execution time.

2. Variables are set up in a table in order of their first appearance in the program. Later in the program, BASIC searches the table for the variable at each reference. Variables at the head of the table take less time to search for than those at the end. Therefore, reuse variable names and keep the list of variables as short as possible.

3. Use NEXT without the index variable.

4. Use integer variables whenever possible.

5. Use variables instead of constants, expecially in FOR loops and other code that must be executed repeatedly.

APPENDIX I:

CONVERTING BASIC PROGRAMS NOT WRITTEN FOR SYSTEM 50 or SYSTEM 70

Though implementations of BASIC on different computers are in many ways similar, there are some incompatibilities between ADDS* BASIC™ and the BASIC used on other computers.

1. Strings

A number of BASICs require the length of strings to be declared before they are used. In some of these BASICs a declaration of the form DIM A$(I,J) declares a string array of J elements each of which has a length I. All dimension statements of this type should be removed from the program. In ADDS*BASIC™, the statement DIM A$(J) declares a string array of J elements each of which has a length up to 255 characters. The SPACE$ function (see Section 5) may be used to declare the length of each element of the string array.

OLD	NEW
DIM A$(I,J)	DIM A$(I) or DIM A$(I) FOR I=1 TO N A$(I)=SPACE$(J) NEXT I

ADDS*BASIC™ uses the plus sign for the string concatenation, not the comma or ampersand.

OLD	NEW
Y$=A$,B$ Y$=A$&B$	Y$=A$+B$

ADDS*BASIC™ uses LEFT$, RIGHT$ and MID$ to take substrings of strings. Some other BASICs use A$(I) to access the Ith character of the string A$, and A$(I,J) to take a substring of A$ from character position I to character position J. Convert as follows:

OLD	NEW
A$(I)	MID$(A$,I,1)
A$(I,J)	MID$(A$,I,J-I+1)

This assumes that the reference to a subscript of A$ is in an expression or is on the right side of an assignment. If the reference to A$ is on the left-hand side of an assignment, and X$ is the string expression used to replace characters in A$, convert as follows:

OLD	NEW
A$(I)=X$	MID$(A$,I,1)=X$
A$(I,J)=X$	MID$(A$,I,J-I+1)=X$

2. Multiple Assignments

Some BASICs allow statements of the form:

```
500 LET B=C=0
```

This statement would set the variables B and C to zero. In ADDS*BASIC™ this has an entirely different effect. All the "=" signs to the right of the first one would be interpreted as logical comparison operators. This would set the variable B to -1 if C equalled 0. If C did not equal 0, B would be set to 0. The easiest way to convert statements like this one is to rewrite them as follows:

```
500 C=0:B=C
```

3. Multiple Statements on a Single Line

Some BASICs use a backslash (\) instead of a colon (:) to delimit multiple statements on a line. Change each backslash in the program to a colon.

4. Matrix Functions

Programs which use the MAT functions available in some BASICs will have to be re-written using FOR...NEXT loops to perform the appropriate operations.

The PRINT FORM switch and certain keys which have particular use and significance under ADDS*BASIC™ are described here.

PRINT FORM SWITCH	In the ON position, this switch causes the Status Line to act as a single line CRT screen. Scrolling of the previous 24 lines of the screen is inhibited.
ENTER	The ENTER key must be depressed after each BASIC line is typed to enter the line in memory.
NEW LINE	The NEW LINE key breaks a BASIC line into shorter parts and results in a single BASIC statement on multiple lines.
CTRL	The CTRL key is used in conjunction with other keys to perform special functions. These are described below.
CTRL-Ø	These keys are depressed during start-up to load the disk operating system.
CTRL-A	Depression of these keys permits the user to recover a previously typed direct statement. Depressing CTRL-A causes the system to enter the EDIT mode. The user may then use the INS CHAR key to add a line number.
CTRL-C	Depression of these keys interrupts the execution of the current program or direct statement. The system returns to the command level of BASIC and responds with Ok.
CTRL-O	Depression of these keys suppresses all output until an INPUT statement is encountered, another CTRL-O is typed, an error occurs or BASIC returns to command level.
CTRL-S	Depression of these keys causes program execution to pause. Depressing any key causes execution to resume.
RESET	A single depression (ADOS Release 1.3a and below) or a double depression (ADOS Release 1.4 and above) of the RESET key causes the current sequence to be aborted, and returns control to the **Command Processor** after rebooting the system. However, it is good programming practice to use the SYSTEM statement instead of the RESET key to return control to the **Command Processor**. The SYSTEM statement closes all files. With the RESET key, open data files will be lost.

Important Note: After switching disks on any drive, depress the RESET key or enter the RESET statement to construct a new map of the directory.

Indicates current line for EDIT, RENUM, DELETE, LIST, LLIST statements.