# OS-CP/M®

# BASIC

Sept. 1978

# Microsoft CP/M BASIC

## Addendum to Microsoft BASIC Manual for Users of CP/M Operating Systems

A CP/M version of BASIC (ver 4.5) is now available from Microsoft. This version of BASIC is supplied on a standard size 3740 single density diskette. The name of the file is MBASIC.COM. To run MBASIC, bring up CP/M and type the following:

```
A>MBASIC <carriage return>
```

The system will reply:

```
xxxx Bytes Free
BASIC Version 4.5
(CP/M Version)
Copyright 1977  (C) by Microsoft
Ok
```

You are now ready to use MBASIC. MBASIC is identical to Altair Disk BASIC version 4.1, with the following exceptions:

1. MBASIC requires 17K of memory.(A 28K or larger CP/M system is recommended).

2. The initialization dialog has been replaced by a set of options which are placed after the MBASIC command to CP/M. The format of the command line is:

   ```
   A>MBASIC [<filename>] [/F:<number of files>]
           [/M:<highest memory location>]
   ```

   Items enclosed in brackets are optional.

   If <filename> is present, MBASIC proceeds as if a RUN <filename> command were typed after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than 9 characters long. This allows BASIC programs to be executed in batch mode using the SUBMIT facility of CP/M. Such programs should include a SYSTEM statement (see below) to return to CP/M when they have finished, allowing the next program in the batch stream to execute.

   If /F:<number of files> is present, it 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 MBASIC. In some cases it is desirable to set the amount of memory well below the CP/M's FDOS to reserve space for assembly language subroutines. In all cases, <highest memory location> should be below the start of FDOS (whose address is contained in locations 6 and 7). If the /M option is omitted, all memory up to the start of FDOS is used.

NOTE

Both <number of files> and <highest memory location> are numbers that may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

Examples:

| | |
|---|---|
| A>MBASIC PAYROLL.BAS | Use all memory and 3 files, load and execute PAYROLL.BAS. |
| A>MBASIC INVENT/F:6 | Use all memory and 6 files, load and execute INVENT.BAS. |
| A>MBASIC /M:32768 | Use first 32K of memory and 3 files. |
| A>MBASIC DATACK/F:2/M:&H9000 | Use first 36K of memory, 2 files, and execute DATACK.BAS |

3. The DSKF function is not supported by MBASIC. Use CP/M STAT.

4. The FILES statement in MBASIC takes the form FILES[<filename>] . If <filename> is omitted, all the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the file name or extension will match any file or any extension.

   Examples:

```
FILES
FILES "*.BAS"
FILES "B:*.*
FILES "TEST?.BAS"
```

5. The LOF(x) function returns the number of records present in the last extent read or written (usually by a PUT or GET).

6. CSAVE and CLOAD are not implemented.

7. LLIST and LPRINT assume a 132 character wide printer and write their output to the CP/M LST: device.

8. All filenames may include A: or B: as the first two characters to specify a disk drive, otherwise the currently selected drive is used.

9. Filenames themselves follow the normal CP/M naming conventions.

10. A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN <filename> commands if no "." appears in the filename and the filename is less than nine characters long.

11. The error messages "DISK NOT MOUNTED", "DISK ALREADY MOUNTED", "OUT OF RANDOM BLOCKS", and "FILE LINK ERROR" are not included in MBASIC.

12. The CONSOLE statement is not included.

13. To return to CP/M use the SYSTEM command or statement. SYSTEM closes all files and then performs a CP/M warm start. Control-C always returns to MBASIC, not to CP/M.

14. If you wish to change diskettes during MBASIC operation, use RESET. RESET closes all files and then forces CP/M to re-read all diskette directory information. Never remove diskettes while running MBASIC unless you have given a RESET command. The RESET statement takes the place of the MOUNT and UNLOAD statements in Altair BASIC.

15. MBASIC will operate properly on both Z-80 and 8080 systems.

16. MBASIC does not use any of the restart (RST) instruction vectors.

17. The FRCINT routine is located at 103 hex and the MAKINT routine at 105 hex (add 1000 hex for ADDS versions). These routines are used to convert the argument to an integer for assembly language subroutines.

18. If the LEFT$ or RIGHT$ string functions have zero as the number of characters argument, they will return the null (length zero) string.

19. The ERR() Disk error function is not supported as CP/M handles all disk error recovery.

20. Control-H (backspace) deletes the last character typed and is echoed to the terminal.

21. RESTORE <line number> may now be used to set the DATA pointer to a specific line.

22. All error messages and prompts are printed with lower case characters when appropriate.

23. Control-S may be used to cause program execution to pause. In the suspended execution state, control-C will cause a return to BASIC's command level, and any other character will cause the program to resume execution.

24. The EOF function may be used with random files. If a GET is done past end of file, EOF will = -1. This may be used to find the size of a file using a binary search or other algorithm.

25. LSET/RSET may be used on any string. The previous restriction to FIELDed strings has been eliminated.

26. The string function INPUT$(<number of characters> [,[#]<file number>]) may be used to read <number of characters> from either the console or a disk file. If the console is used for input, no characters will be echoed and all control characters are passed through except Control-C, which is used to interrupt execution of the INPUT$ function.

27. VARPTR(#<file number>) returns the address of the disk data buffer for file <file number>.

Commands:

| | | | | |
|---|---|---|---|---|
| AUTO | CLEAR | CONT | DELETE | EDIT |
| FILES | LIST | LLIST | LOAD | MERGE |
| NEW | NULL | RENUM | RESET | RUN |
| SAVE | SYSTEM | TRON | TROFF | WIDTH |

Program Statements:

| | | | | |
|---|---|---|---|---|
| DEFNx | DEFDBL | DEFINT | DEFSNG | DEFSTR |
| DIM | END | ERASE | ERROR | FOR |
| GOSUB | GOTO | IF..THEN[ELSE] | LET | NEXT |
| ON...ERROR | ON...GOSUB | ON...GOTO | OUT | POKE |
| REM | RESUME | RETURN | STOP | SWAP |
| WAIT | | | | |

Input/Output Statements:

| | | | | |
|---|---|---|---|---|
| CLOSE | DATA | FIELD | GET | INPUT |
| KILL | LINEINPUT | LSET | NAME | OPEN |
| PRINT | PUT | READ | RESTORE | RSET |

Operators:

| | | | | |
|---|---|---|---|---|
| = | - | + | * | / |
| ^ | \ | MOD | NOT | AND |
| OR | XOR | IMP | EQV | < |
| > | <= | >= | <> | |

Arithmetic Functions:

| | | | | |
|---|---|---|---|---|
| ABS | ATN | CDBL | CINT | COS |
| CSNG | ERL | ERR | EXP | FRE |
| INP | INT | LOG | LPOS | PEEK |
| POS | RND | SGN | SIN | SPC |
| SQR | TAB | USRn | VARPTR | |

String Functions:

| | | | | |
|---|---|---|---|---|
| ASC | CHR$ | FRE | HEX$ | INSTR |
| LEFT$ | LEN | MID$ | OCT$ | RIGHT$ |
| SPACE$ | STRING$ | STR$ | VAL | |

Input/Output Functions:

| | | | | |
|---|---|---|---|---|
| CVD | CVI | CVS | EOF | LOC |
| LOF | MKD$ | MKI$ | MKS$ | |

Microsoft Disk BASIC also supports files on multiple floppy disks:

1. Sequential files with variable length records
2. Random files (record I/O)
3. Complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, NAME, etc.
4. Up to 255 files per floppy disk
5. Runs standalone or under CP/M or ISIS-II operating systems

# Microsoft BASIC

# Reference Manual

# Microsoft BASIC

## Overview

Microsoft BASIC is an extensive implementation of BASIC for 8080 and Z-80 microprocessors. Its features are comparable to those of BASICs found on minicomputers and large mainframes.

## Current Versions of Microsoft BASIC

Microsoft BASIC is currently in its fourth major release (4.3). Each release consists of four different versions of BASIC:

1. 4K version: Stripped down version to run in minimum memory. Includes direct statement execution, dynamic dimensioning of arrays and multiple statements per line.

2. 8K version: Standard version. Includes string manipulation and multiple dimension arrays. (Also available for 6800 and 650x series MPUs.)

3. Extended version: Requires 16K of memory. Features include integers, double precision, EDIT, AUTO, RENUM, PRINT USING, etc.

4. Disk version: Requires 20K of memory. All features of Extended version plus random and sequential file access on floppy disk.

The different versions are generated from the same source files using conditional assembly switches. Each version is upward compatible with larger versions.

5. Page 59, last line:

```
520 CLOSE #1
```

CHANGE TO:

```
520 CLOSE 1
```

6. Page 70, CLEAR [<expression>] explanation:

   Same as CLEAR but sets string space to the value . . .

   CHANGE TO:

   Same as CLEAR but sets string space (see 4-1) to the value . . .

7. Page 70, CLOAD <string expression> explanation, second line:

   . . . character of STRING expression> to be . . .

   CHANGE TO:

   . . . character of <STRING expression> to be . . .

8. Page 71:

   CSAVE*<array name>            8K (cassette), Disk

   CHANGE TO:

   CSAVE*<array name>            8K (cassette), Extended, Disk

9. Page 75. Insert the following after LET and before LPRINT.

   ADDITION:

   LINE INPUT  LINE INPUT "prompt string"; string variable name

   Extended, Disk

   LINE INPUT prints the prompt string on the terminal and assigns all input from the end of the prompt string to the carriage return to the named string variable. No other prompt is printed if the prompt string is omitted. LINE INPUT may not be edited by Control/A.

10. Page 76, POKE explanation, second line:

    . . . If I is negative, address is 65535+I, . . .

    CHANGE TO:

    . . . If I is negative, address is 65536+I, . . .

# BASIC Reference Manual

## Addenda, April, 1977

1. Page 33, sub-paragraph b:

   LINE INPUT ["<prompt string>",]; <string variable name>

   CHANGE TO:

   LINE INPUT ["<prompt string>";] <string variable>

2. Page 40, Paragraph 5-3b, line 9:

   The of the <integer expression> is the starting address of . . .

   CHANGE TO:

   The <integer expression> is the starting address of . . .

3. Page 41. Insert the following paragraphs between Paragraphs 3 and 4.

   ADDITION:

   The string returned by a call to USR with a string argument is that string the user's routine sets up in the descriptor. Modifying [D,E] does not affect the returned string. Therefore, the statement:

   ```
   C$=USR(A$)
   ```

   results in A$ also being set to the string assigned to C$. To avoid modifying A$ in this statement, we would use:

   ```
   C$=USR(A$+" ")
   ```

   so that the user's routine modifies the descriptor of a string temporary instead of the descriptor for A$.

   A string returned by a user's routine should be completely within the bounds of the storage area used by the original string. Increasing a string's length in a user routine is guaranteed to cause problems.

4. Page 49, last paragraph, line 7:

   . . . leading $ signs, nor can negative numbers be output unless the sign is forced to be trailing.

   CHANGE TO:

   . . . leading $ signs.

11. Page 80, OCT$:

    OCT$ OCT$(X) 8K, Extended, Disk

    CHANGE TO:

    OCT$ OCT$(X) Extended, Disk

12. Page 81:

    SPACE$ SPACE$(I) 8K, Extended, Disk

    CHANGE TO:

    SPACE$ SPACE$(I) Extended, Disk

13. Page 91, line 4:

    . . . question (see Appendix E).

    CHANGE TO:

    . . . question (see Appendix H).

14. Page 95, first paragraph, line 3:

    . . . For instructions on loading Disk BASIC, see Appendix E.

    CHANGE TO:

    . . . For instructions on loading Disk BASIC, see Appendix H.

15. Page 103, line 11:

    C (in extended) retains CONSOLE function.

    CHANGE TO:

    C (in Extended and Disk) retains CONSOLE and all other functions.

16. Page 112, Paragraph 4, Line 3:

    USRLOC for 4K and 8K Altair BASIC version 4.0 is 111 decimal.

    CHANGE TO:

    USRLOC for 4K and 8K Altair BASIC version 4.0 is 111 octal.

17. Page 114, third paragraph, line 2:

    . . . by the first character of the STRING expression>.

    CHANGE TO:

. . . by the first character of the <string expression>. Note that the program named A is saved by CSAVE"A".

18. Index, line 12:

ADDITION:

NULL . . . . . . . . . . . . . . . . . . 72

## CONTENTS

# 1. SOME INTRODUCTORY REMARKS

## 1-1 Introduction to this Manual.

a. Conventions. For the sake of simplicity, some conventions will be followed in discussing the features of the Altair BASIC language.
1. Words printed in capital letters must be written exactly as shown. These are mostly names of instructions and commands.
2. Items enclosed in angle brackets (<>) must be supplied as explained in the text. Items in square brackets ([]) are optional. Items in both kinds of brackets, [<W>], for example, are to be supplied if the optional feature is used. Items followed by dots (...) may be repeated or deleted as necessary.
3. Shift/ or Control/ followed by a letter means the character is typed by holding down the Shift or Control key and typing the indicated letter.
4. All indicated punctuation must be supplied.

b. Definitions. Some terms which will become important are as follows:

Alphanumeric character: all letters and numerals taken together are called alphanumeric characters.

Carriage Return: Refers both to the key on the terminal which causes the carriage, print head or cursor to move to the beginning of the next line and to the command that the carriage return key issues which terminates a BASIC line.

Command Level: After Altair BASIC prints OK, it is at the command level. This means it is ready to accept commands.

Commands and Statements: Instructions in Altair BASIC are loosely divided into two classes, Commands and Statements. Commands are instructions normally used only in direct mode (see Modes of Operation, section 1-2). Some commands, such as CONT, may only be used in direct mode since they have no meaning as program statements. Some commands, such as DELETE, are not normally used as program statements because they cause a return to command level. But most commands will find occasional use as program statements. Statements are instructions that are normally used in indirect mode. Some statements, such as DEF, may only be used in indirect mode.

Edit: The process of deleting, adding and substituting lines in a program and that of preparing data for output according to a predetermined format will both be referred to as "editing." The particular meaning in use will be clear from the context.

Integer Expression: An expression whose value is truncated to an integer. The components of the expression need not be of integer type.

Reserved Words: Some words are reserved by BASIC for use as statements and commands. These are called reserved words and they may not be used in variable or function names.

Special Characters: some characters appear differently on different terminals. Some of the most important of these are the following:

^ (caret) appears on some terminals as ↑ (up-arrow)
~ (tilde) does not appear on some terminals and prints as a blank
_ (underline) appears on some terminals as ← (back-arrow).

String Literal: A string of characters enclosed by quotation marks (") which is to be input or output exactly as it appears. The quotation marks are not part of the string literal, nor may a string literal contain quotation marks. (""HI, THERE""is not legal.)

Type: While the actual device used to enter information into the computer differs from system to system, this manual will use the word "type" to refer to the process of entry. The user types, the computer prints. Type also refers to the classifications of numbers and strings.

## 1-2 Modes of Operation.

Altair BASIC provides for operation of the computer in two different modes. 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 themselves are lost after execution. This mode is useful for debugging and for using Altair BASIC in a "calculator" mode for quick computations which do not justify the design and coding of complete programs.

In the indirect mode, the computer executes instructions from a program stored in memory. Program lines are entered into memory if they are preceded by a line number. Execution of the program is initiated by the RUN

In the indirect mode, the computer executes instructions from a program stored in memory. Program lines are entered into memory if they are preceded by a line number. Execution of the program is initiated by the RUN commands.

## 1-3 Formats.

a. Lines. The line is the fundamental unit of an Altair BASIC program. The format for an Altair BASIC line is as follows:

```
     nnnnn <BASIC statement>[:<BASIC statement>...]
```

Each Altair BASIC line begins with a number. The number corresponds to the address of the line in memory and indicates the order in which the statements in the line will be executed in the program. It also provides for branching linkages and for editing. Line numbers must be in the range Ø to 65529. A good programming practice is to use an increment of 5 or 1Ø between successive line numbers to allow for insertions.

1) Line numbers may be generated automatically in the Extended and Disk versions of Altair BASIC by use of the AUTO and RENUM commands. The AUTO command provides for automatic insertion of line numbers when entering program lines. The format of the AUTO command is as follows:

```
     AUTO[<initial line>[,[<increment>]]
Example;
     AUTO 1ØØ,1Ø
     1ØØ INPUT X,Y
     11Ø PRINT SQR(X^2+Y^2)
     12Ø ^C
     OK
```

AUTO will number every input line until Control/C is typed. If the <initial line> is omitted, it is assumed to be 1Ø and an increment of 1Ø is assumed if <increment> is omitted. If the <initial line> is followed by a comma but no increment is specified, the increment last used in an AUTO statement is assumed.

If AUTO generates a line number that already exists in the program currently in memory, it prints the number followed by an asterisk. This is to warn the user that any input will replace the existing line.

2) The RENUM command allows program lines to be "spread out" so that a new line or lines may be inserted between existing lines. The format of the RENUM command is as follows:

```
RENUM [<NN>[<MM>[,<II>]]]
```

where NN is the new number of the first line to be resequenced. If omitted, NN is assumed to be 1Ø. Lines less than MM will not be renumbered. If MM is omitted, the whole program will be resequenced. II is the increment between the lines to be resequenced. If II is omitted, it is assumed to be 1Ø. Examples:

RENUM Renumbers the whole program to start at line 1Ø with an increment of 1Ø between the new line numbers.

RENUM 1ØØ,,1ØØ Renumbers the whole program to start at line 1ØØ with an increment of 1ØØ.

RENUM 6ØØØ,5ØØØ,1ØØØ Renumbers the lines from 5ØØØ up so they start at 6ØØØ with an increment of 1ØØØ.

NOTE

RENUM cannot be used to change the order of program lines (for example, RENUM 15,3Ø when the program has three lines numbered 1Ø, 2Ø and 3Ø) nor to create line numbers greater than 65529. An ILLEGAL FUNCTION CALL error will result.

All line numbers appearing after a GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL<relational operator> will be properly changed by RENUM to reference 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" will be printed. This line reference (XXXXX) will not be changed by RENUM, but line number YYYYY may be changed.

3) In the Extended and Disk versions, the current line number may be designated by a period (.) anywhere a line number reference is required. This is particularly useful in the use of the EDIT command. See section 5-4.

4) Following the line number, one or more BASIC statements are written. The first word of a statement identifies the operations to be performed. The list of arguments which follows the identifying word serves several purposes. It can contain (or refer symbolically to) the

data which is to be operated upon by the statement. In some important instructions, the operation to be performed depends upon conditions or options specified in the list.

Each type of statement will be considered in detail in sections 2, 3 and 4.

More than one statement can be written on one line if they are separated by colons (:). Any number of statements can be joined this way provided that the line is no more than 72 characters long in the 4K and 8K versions, or 255 characters in the Extended and Disk versions. In the Extended and Disk versions, lines may be broken with the LINE FEED key. Example:

```
100 IF X<Y+37<line feed>
    THEN 5 <line feed>
    ELSE PRINT(X)<carriage return>
```

The line is shown broken into three lines, but it is input as one BASIC line.

b. REMarks. In many cases, a program can be more easily understood if it contains remarks and explanations as well as the statements of the program proper. In Altair BASIC, the REM statement allows such comments to be included without affecting execution of the program. The format of the REM statement is as follows:

```
 REM <remarks>
```

A REM statement is not executed by BASIC, but branching statements may link into it. REM statements are terminated by the carriage return or the end of the line but not by a colon. Example:

```
100 REM DO THIS LOOP:FOR I=1TO10       -the FOR statement
                                       will not be executed
101 FOR I=1 TO 10: REM DO THIS LOOP    -this FOR statement will
                                       be executed.
```

In Extended and Disk versions, remarks may be added to the end of a program line separated from the rest of the line by a single quotation mark ('). Everything after the single quote will be ignored.

c. Errors. When the BASIC interpreter detects an error that will cause the program to be terminated, it prints an error message. The error message formats in Altair BASIC are as follows:

```
Direct statement     ?XX ERROR
```

Indirect statement    ?XX ERROR IN nnnnn

XX is the error code or message (see section 6-5 for a list of error codes and messages) and 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. These errors will be discussed in the description of the individual statements.

## 1-4 Editing - elementary provisions.

Editing features are provided in Altair BASIC so that mistakes can be corrected and features can be added and deleted without affecting the remainder of the program. If necessary, the whole program may be deleted. Extended and Disk Altair BASIC have expanded editing facilities which will be discussed in section 5.

a. Correcting single characters. If an incorrect character is detected in a line as it is being typed, it can be corrected immediately with the backarrow ( underline on some terminals) or ,except in 4K, the RUBOUT key. Each stroke of the key deletes the immediately preceding character. If there is no preceding character, a carriage return is issued and a new line is begun. Once the unwanted characters are removed, they can be replaced simply by typing the rest of the line as desired.

When RUBOUT is typed, a backslash (\) is printed and then the character to be deleted. Each successive RUBOUT prints the next character to be deleted. Typing a new character prints another backslash and the new character. All characters between the backslashes are deleted.

Example:

```
100 X=\=X\Y=10     Typing two RUBOUTS deleted the '='
                   and 'X' which were subsequently
                   replaced by Y= .
```

b. correcting lines. A line being typed may be deleted by typing an at-sign (@) instead of typing a carriage return. A carriage return is printed automatically after the line is deleted. Except in 4K, typing Control/U has the same effect.

In the Extended and Disk versions, typing Control/A instead of the carriage return will allow all the features of the EDIT command (except the A command) to be used on the line currently being typed. See section 5-4.

c. correcting whole programs. The NEW command causes the entire current program and all variables to be deleted. NEW is generally used to clear memory space preparatory to entering a new program.

## 2. STATEMENTS AND EXPRESSIONS.

### 2-1. Expressions.

The simplest BASIC expressions are single constants, variables and function calls.

a. Constants. Altair BASIC accepts integers or floating point real numbers as constants. All but the 4K version of Altair BASIC accept string constants as well. See section 4-1. Some examples of acceptable numeric constants follow:

```
123
3.141
Ø.Ø436
1.25E+Ø5
```

Data input from the terminal or numeric constants in a program may have any number of digits up to the length of a line (see section 1-3a). In 4K and 8K Altair BASIC, however, only the first 7 digits of a number are significant and the seventh digit is rounded up. Therefore, the command

```
PRINT 1.23456789Ø123
```

produces the following output:

```
1.23457
OK
```

In Extended and Disk versions of Altair BASIC, double precision format allows 17 significant digits with the 17th digit rounded up.

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.

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

3. If the absolute value of the number is greater than or equal to .Ø1 and less than or equal to 999999, it is printed in fixed point notation with no exponent.

4. In Extended and Disk versions, fixed point values up to 9999999999999999 are possible.

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

The formats of scientific notation are as follows:

```
SX.XXXXXESTT              single precision

SX.XXXXXXXXXXXXXXXDSTT    double precision
```

where S stands for the signs of the mantissa and the exponent (they need not be the same, of course), X for the digits of the mantissa and T for the digits of the exponent. E and D may be read "...times ten to the power...." Non-significant zeros are suppressed in the mantissa, but two digits are always printed in the exponent. The sign convention in rule 1 is followed for the mantissa. The exponent must be in the range -38 to +38. The largest number that may be represented in Altair BASIC is 1.7Ø141E38, the smallest positive number is 2.9387E-38. The following are examples of numbers as input and as output by Altair BASIC:

| Number | Altair BASIC Output |
|---|---|
| +1 | 1 |
| -1 | -1 |
| 6523 | 6523 |
| 1E2Ø | 1E2Ø |
| -12.34567E-1Ø | -1.23456E-Ø9 |
| 1.234567E-7 | 1.23457E-Ø7 |
| 1ØØØØØØ | 1E+Ø6 |
| .1 | .1 |
| .Ø1 | .Ø1 |
| .ØØØ123 | 1.23E-Ø4 |
| -25.46Ø | -25.46 |

The Extended and Disk versions of Altair BASIC allow numbers to be represented in integer, single precision or double precision form. The type of a number constant is determined according to the following rules:

1. A constant with more than 7 digits or a 'D' instead of 'E' in the exponent is double precision.

2. A constant outside the range -32768 to 32767 with 7 or fewer digits and a decimal point or with an 'E' exponent is single precision.

3. A constant in the range -32768 to 32767 and no decimal point is integer.

4. A constant followed by an exclamation point (!) is single precision; a constant followed by a pound sign (#) is double precision.

Two additional types of constants are allowed in Extended and Disk versions of Altair BASIC. Hexadecimal (base sixteen) constants may be explicitly designated by the symbol &H preceding the number. The constant may not contain any characters other than the digits Ø - 9 or letters A - F, or a SYNTAX ERROR will occur. Octal constants may be designated either by &O or just the & sign.

In all formats, a space is printed after the number. In all but the 4K version, Altair BASIC checks to see if the entire number will fit on the current line. If not, it issues a carriage return and prints the whole number on the next line.

b. Variables

1) A variable represents symbolically any number which is assigned to it. The value of a variable may be assigned explicitly by the programmer or may be assigned as the result of calculations in a program. Before a variable is assigned a value, its value is assumed to be zero. In 4K , a variable name consists of one or two characters. The first character is any letter. The second character must be a numeral. In other versions of Altair BASIC, the variable name may be any length, but any alphanumeric characters after the first two are ignored. The first character must be a letter. No reserved words may appear as variable names or within variable names. The following are examples of legal and illegal Altair BASIC variables:

| Legal | Illegal |
|---|---|
| In 4K and 8K Altair BASIC: | |
| A | %A (first character must be alphabetic.) |
| Z1 | Z1A (variable name is too long for 4K) |

Other versions:

| | |
|---|---|
| TP | TO (variable names cannot be reserved words) |
| PSTG$ | |
| COUNT | RGOTO (variable names cannot contain reserved words.) |

In all but 4K Altair BASIC, a variable may also represent a string. Use of this feature is discussed in section 4.

2) Extended and Disk versions of Altair BASIC allow the use of Integer and Double Precision variables as well as Single Precision and Strings. The type of a variable may be explicitly declared in Extended and Disk versions of Altair BASIC by using one of the symbols in the table below as the last character of the variable name.

| Type | Symbol |
|---|---|
| Strings (Ø to 255 characters) | $ |
| Integers (-32768 to 32767) | % |
| Single Precision (up to 7 digits, exponent between -38 and +38) | ! |
| Double Precision (up to 16 digits, exponent between -38 and +38) | # |

Internally, BASIC handles all numbers in binary. Therefore, some 8 digit single precision and 17 digit double precision numbers may be handled correctly. If no type is explicitly declared, type is determined by the first letter of the variable name according to the type table. The table of types may be modified with the following statements.

| | |
|---|---|
| DEFINT r | Integer |
| DEFSTR r | String |
| DEFSNG r | Single Precision |
| DEFDBL r | Double Precision |

where r is a letter or range of letters to be designated. Examples:

| | |
|---|---|
| 15 DEFINT I-N | Variable names beginning with the letters I-N are to be of integer type. |
| 2Ø DEFDBL D | Variable names beginning with D are to be of double precision type. |

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

3) Integer variables should be used wherever possible since they take the least amount of space in memory and integer arithmetic is much faster than single precision arithmetic.

Care must be exercised when single precision and double precision numbers are mixed. Since single precision numbers can have more significant digits than will be printed, a double precision variable set to a single precision value may not print the same as the single precision variable.

```
10 A=1.01                          single precision value
20 B#=A*10:C#=CDBL(A)*10#          convert to double precision
30 PRINTA;B#;C#;CDBL(A)            in various ways
RUN
 1.01  10.10000038146973  10.09999990463257  1.009999990463257
OK
```

In order to assure that double precision numbers will print the same as single precision, the VAL and STR$ functions should be used. For example:

```
10 A=1.01
20 B#=VAL(STR$(A)):C#=B#*10#
30 PRINT A;B#;C#
RUN
 1.01  1.01  10.1
OK
```

c. Array Variables. It is often advantageous to refer to several variables by the same name. In matrix calculations, for example, the computer handles each element of the matrix separately, but it is convenient for the programmer to refer to the whole matrix as a unit. For this purpose, Altair BASIC provides subscripted variables, or arrays. The form of an array variable is as follows:

```
VV(<subscript>[,<subscript>...])
```

where VV is a variable name and the subscripts are integer expressions. Subscripts may be enclosed in parentheses or square brackets. An array variable may have only one dimension in 4K, but in all other versions of Altair BASIC it may have as many dimensions as will fit on a single line. The smallest subscript is zero. Examples:

| | |
|---|---|
| A(5) | The sixth element of array A. The first element is A(0). |
| ARRAY(I,2*J) | The address of this element in a two-dimensional array is determined by evaluating the expressions in parentheses at the time of the reference to the |

array and truncating to integers. If I=3 and J=2.4, this refers to ARRAY(3,4).

The DIM statement allocates storage for array variables and sets all array elements to zero. The form of the DIM statement is as follows:

```
DIM VV(<subscript>[,<subscript>...])
```

where VV is a legal variable name. Subscript is an integer expression which specifies the largest possible subscript for that dimension. Each DIM statement may apply to more than one array variable. Some examples follow:

```
113 DIM A(3), D$(2,2,2)
114 DIM R2%(4), B(10)
115 DIM Q1(N), Z#(2+I)
```

Arrays may be dimensioned dynamically during program execution. At the time the DIM is executed, the expression within the parentheses is evaluated and the results truncated to integer.

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. A BS or SUBSCRIPT OUT OF RANGE error message will be issued if an attempt is made to reference an array element which is outside the space allocated in its associated DIM statement. This can occur when the wrong number of dimensions is used in an array element reference. For example:

```
30 LET A(1,2,3)=X  when A has been dimensioned by
10 DIM A(2,2)
```

A DD or REDIMENSIONED ARRAY error occurs when a DIM statement for an array is found after that array has been dimensioned. This often occurs when a DIM statement appears after an array has been given its default dimension of 10.

d. Operators and Precedence. Altair BASIC provides a full range of arithmetic and (except in 4K) logical operators. The order of execution of operations in an expression is always according to their precedence as shown in the table below. The order can be specified explicitly by the use of parentheses in the normal algebraic fashion.

<u>Table of Precedence</u>

Operators are shown here in decreasing order of precedence. Operators listed in the same entry in the table have the same precedence and are executed in order from left to right in an expression.

1. Expressions enclosed in parentheses ()

2. ^ exponentiation (not in 4K). Any number to the zero power is 1. Zero to a negative power causes a /Ø or DIVISION BY ZERO error.

3. - negation, the unary minus operator

4. *,/ multiplication and division

5. \ integer division (available in Extended and Disk versions, see section 5-2)

6. MOD (available in Extended and Disk versions. See section 5-2)

7. +,- addition and subtraction

8. relational operators

```
          = equal
          <> not equal
          < less than
          > greater than
          <=,=< less than or equal to
          >=,=> greater than or equal to
```

(the logical operators below are not available in 4K)

9. NOT logical, bitwise negation

1Ø. AND logical, bitwise disjunction

11. OR logical, bitwise conjunction

(The logical operators below are available only in Extended and Disk versions.)

12. XOR logical, bitwise exclusive OR

13. EQV logical, bitwise equivalence

14. IMP logical, bitwise implication

In 4K Altair BASIC, relational operators may be used only once in an IF statement. In all other versions, relational operators may be used in any expressions. Relational expressions have the value either of True (-1) or False (Ø).

e. Logical Operations. Logical operators may be used for bit manipulation and Boolean algebraic functions. The AND, OR, NOT, XOR, EQV and IMP operators convert their arguments into sixteen bit, signed, two's complement integers in the range -32768 to 32767. After the operations are performed, the result is returned in the same form and range. If the arguments are not in this range, an FC or ILLEGAL FUNCTION CALL error message will be printed and execution will be terminated. Truth tables for the logical operators appear below. The operations are performed bitwise, that is, corresponding bits of each argument are examined and the result computed one bit at a time. In binary operations, bit 7 is the most significant bit of a byte and bit Ø is the least significant.

AND

| X | Y | X AND Y |
|---|---|---|
| 1 | 1 | 1 |
| 1 | Ø | Ø |
| Ø | 1 | Ø |
| Ø | Ø | Ø |

OR

| X | Y | X OR Y |
|---|---|---|
| 1 | 1 | 1 |
| 1 | Ø | 1 |
| Ø | 1 | 1 |
| Ø | Ø | Ø |

NOT

| X | NOT X |
|---|---|
| 1 | Ø |
| Ø | 1 |

XOR

| X | Y | X XOR Y |
|---|---|---|
| 1 | 1 | Ø |
| 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 |

IMP

| X | Y | X IMP Y |
|---|---|---|
| 1 | 1 | 1 |
| 1 | Ø | Ø |
| Ø | 1 | 1 |
| Ø | Ø | 1 |

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

| | |
|---|---|
| 63 AND 16=16 | 63=binary 111111 and 16=binary 1ØØØØ, so 63 AND 16=16 |
| 15 AND 14=14 | 15= binary 1111 and 14=binary 111Ø, so 15 AND 14=binary 111Ø=14. |
| -1 AND 8=8 | -1=binary 1111111111111111 and 8=binary 1ØØØ, so -1 AND 8=8. |
| 4 OR 2=6 | 4=binary 1ØØ and 2=binary 1Ø so 4 OR 2=binary 11Ø=6. |
| 1Ø OR 1Ø=1Ø | binary 1Ø1Ø OR'd with itself is 1Ø1Ø=1Ø. |
| -1 OR -2=-1 | -1=binary 1111111111111111 and -2=111111111111111Ø, so -1 OR -2=-1. |
| NOT Ø=-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. |

A typical use of logical operations is 'masking', testing a binary number for some predetermined pattern of bits. Such numbers might come from the computer's input ports and would then reflect the condition of some external device. Further applications of logical operations will be considered in the discussion of the IF statement.

f. The LET statement. The LET statement is used to assign a value to a variable. The form is as follows:

```
LET <VV>=<expression>
```

where VV is a variable name and the expression is any valid Altair BASIC arithmetic or, except in 4K, logical or string expression. Examples:

```
1ØØØ LET V=X
11Ø LET I=I+1       the '=' sign heremeans 'is replaced
                    by ....'
```

The word LET in a LET statement is optional, so algebraic equations such as:

```
12Ø V=.5*(X+2)
```

are legal assignment statements.

A SN or SYNTAX ERROR message is printed when BASIC detects incorrect form, illegal characters in a line, incorrect punctuation or missing parentheses. An OV or OVERFLOW error occurs when the result of a calculation is too large to be represented by Altair BASIC's number formats. All numbers must be within the range 1E-38 to 1.7Ø141E38 or -1E-38 to -1.7Ø141E38. An attempt to divide by zero results in the /Ø or DIVISION BY ZERO error message.

For a discussion of strings, string variables and string operations, see section 4.

## 2-2. Branching, Loops and Subroutines.

a. Branching. In addition to the sequential execution of program lines, BASIC provides for changing the order of execution. This provision is called branching and is the basis of programmed decision making and loops. The statements in Altair BASIC which provide for branching are the GOTO, IF...THEN and ON...GOTO statements.

1) GOTO is an unconditional branch. Its form is as follows:

```
GOTO<mmmmm>
```

After the GOTO statement is executed, execution continues at line number mmmmm.

2) IF...THEN is a conditional branch. Its form is as follows:

```
IF<expression>THEN<mmmmm>
```

where the expression is a valid arithmetic, relational or, except in 4K, logical expression and mmmmm is a line number. If the expression is evaluated as non-zero, BASIC continues at line mmmmm. Otherwise, execution resumes at the next line after the IF...THEN statement.

An alternate form of the IF...THEN statement is as follows:

```
IF<expression>THEN<statement>
```

where the statement is any Altair BASIC statement. Examples:

1Ø IF A=1Ø THEN 4Ø If the expression A=1Ø is true, BASIC branches to line 4Ø. Otherwise, execution proceeds at the next line.

15 IF A<B+C OR X THEN 1ØØ The expression after IF is evaluated and if the value of the expression is non-zero, the statement branches to line 1ØØ.

Otherwise, execution continues on the next line.

20 IF X THEN 25 If X is not zero, the statement branches to line 25.

30 IF X=Y THEN PRINT X If the expression X=Y is true (its value is non-zero), the PRINT statement is executed. Otherwise, the PRINT statement is not executed. In either case, execution continues with the line after the IF...THEN statement.

35 IF X=Y+3 GOTO 39 Equivalent to the corresponding IF...THEN statement, except that GOTO must be followed by a line number and not by another statement.

Extended and Disk versions of Altair BASIC provide an expanded IF...THEN statement of the form

```
IF<expression>THEN<YY>ELSE<ZZ>
```

where YY and ZZ are valid line numbers or Altair BASIC statements. Examples:

```
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.

```
IF X=2*Y THEN 5 ELSE PRINT "ERROR"
```

If the expression X=2*Y is true, BASIC branches to line 5; otherwise, the PRINT statement is executed. Extended and Disk Altair BASIC allow a comma before THEN.

IF statements may be nested in the Extended and Disk versions. Nesting is limited only by the length of the line. 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. If a line does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. Example:

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

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

3) ON...GOTO (not in 4K) provides for another type of conditional branch. Its form is as follows:

```
ON<expression>GOTO<list of line numbers>
```

After the value of the expression is truncated to an integer, say I, the statement causes BASIC to branch to the line whose number is Ith in the list. The statement may be followed by as many line numbers as will fit on one line. If I=Ø or is greater than the number of lines in the list, execution will continue at the next line after the ON...GOTO statement. I must not be less than zero or greater than 255, or an FC or ILLEGAL FUNCTION CALL error will result.

b. Loops. It is often desirable to perform the same calculations on different data or repetitively on the same data. For this purpose, Altair BASIC provides the FOR and NEXT statements. The form of the FOR statement is as follows:

```
FOR<variable>=<X>TO<Y>[STEP <Z>]
```

where X,Y and Z are expressions. When the FOR statement is encountered for the first time, the expressions are evaluated. The variable is set to the value of X which is called the initial value. BASIC then executes the statements which follow the FOR statement in the usual manner. When a NEXT statement is encountered, the step Z is added to the variable which is then tested against the final value Y. If Z, the step, 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 BASIC branches back to the statement immediately following the FOR statement. Otherwise, execution proceeds with the statement following the NEXT. If the step is not specified, it is assumed to be 1. Examples:

```
1Ø FOR I=2 TO 11        The loop is executed 1Ø times with
                        the variable I taking on each in-
                        tegral value from 2 to 11.
2Ø FOR V=1 TO 9.3       This loop will execute 9 times un-
                        til V is greater than 9.3
3Ø FOR V=1Ø*N TO 3.4/Z STEP SQR(R)   The initial, final
                        and step expressions need not be
                        integral, but they will be eval-
                        uated only once, before loop-
                        ing begins.
4Ø FOR V=9 TO 1 STEP -1  This loop will be executed 9
                        times.
```

FOR...NEXT loops may be nested. That is, BASIC will execute a FOR...NEXT loop within the context of another loop. An example of two nested loops follows:

```
100 FOR I=1 TO 10
120 FOR J=1 TO I
130 PRINT A(I,J)
140 NEXT J
150 NEXT I
```

Line 130 will print 1 element of A for I=1, 2 for I=2 and so on. If loops are nested, they must have different loop variable names. The NEXT statement for the inside loop variable (J in the example) must appear before that for the outside variable (I). Any number of levels of nesting is allowed up to the limit of available memory.

The NEXT statement is of the form:

```
NEXT[<variable>[,<variable>...]]
```

where each variable is the loop variable of a FOR loop for which the NEXT statement is the end point. In the 4K version, the only form allowed is NEXT with one variable. In all other versions, NEXT without a variable will match the most recent FOR statement. In the case of nested loops which have the same end point, a single NEXT statement may be used for all of them, except in 4K. The first variable in the list must be that of the most recent loop, the second of the next most recent, and so on. If BASIC encounters a NEXT statement before its corresponding FOR statement has been executed, an NF or NEXT WITHOUT FOR error message is issued and execution is terminated.

c. Subroutines. If the same operation or series of operations are to be performed in several places in a program, storage space requirements and programming time will be minimized by the use of subroutines. A subroutine is a series of statements which are executed in the normal fashion upon being branched to by a GOSUB statement. Execution of the subroutine is terminated by the RETURN statement which branches back to the statement after the most recent GOSUB. The format of the GOSUB statement is as follows:

```
GOSUB<line number>
```

where the line number is that of the first line of the subroutine. 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.

Except in the 4K version, subroutines may be branched to conditionally by use of the ON...GOSUB statement, whose form is as follows:

```
ON <expression> GOSUB <list of line numbers>
```

The execution is the same as ON...GOTO except that the line numbers are those of the first lines of subroutines. Execution continues at the next statement after the ON...GOSUB upon return from one of the subroutines.

d. OUT OF MEMORY errors. While nesting in loops, subroutines and branching is not limited by BASIC, memory size limitations restrict the size and complexity of programs. The OM or OUT OF MEMORY error message is issued when a program requires more memory than is available. See Appendix C for an explanation of the amount of memory required to run programs.

## 2-3. Input/Output

a. INPUT. The INPUT statement causes data input to be requested from the terminal. The format of the INPUT statement is as follows:

```
INPUT<list of variables>
```

The effect of the INPUT statement is to cause the values typed on the terminal to be assigned to the variables in the list. When an INPUT statement is executed, a question mark (?) is printed on the terminal signalling a request for information. The operator types the required numbers or strings (or, in 4K, expressions) separated by commas and types a carriage return. If the data entered is invalid (strings were entered when numbers were requested, etc.) BASIC prints 'REDO FROM START?' and waits for the correct data to be entered. If more data was requested by the INPUT statement than was typed, ?? is printed on the terminal and execution awaits the needed data. 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 normally at the statement following the INPUT. Except in 4K, an optional prompt string may be added to an INPUT statement.

```
INPUT["<prompt string>";]<variable list>
```

Execution of the statement causes the prompt string to be printed before the question mark. Then all operations proceed as above. The prompt string must be enclosed in double quotation marks (") and must be separated from the variable list by a semicolon (;). Example:

```
     100 INPUT "WHAT'S THE VALUE";X,Y  causes the following
                              output:

     WHAT'S THE VALUE?
```

The requested values of X and Y are typed after the ? Except in 4K, a carriage return in response to an INPUT statement will cause execution to continue with the values of the variables in the variable list unchanged. In 4K, a SN error results.

b. PRINT. The PRINT statement causes the terminal to print data. The simplest PRINT statement is:

```
     PRINT
```

which prints a carriage return. The effect is to skip a line. The more usual PRINT statement has the following form:

```
     PRINT<list of expressions>
```

which causes the values of the expressions in the list to be printed. String literals may be printed if they are enclosed in double quotation marks (").

The position of printing is determined by the punctuation used to separate the entries in the list. Altair BASIC divides the printing line into zones of 14 spaces each. A comma causes printing of the value of the next expression to begin at the beginning of the next 14 column zone. A semicolon (;) causes the next printing to begin immediately after the last value printed. If a comma or semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line according to the conditions above. Otherwise, a carriage return is printed.

c. DATA, READ, RESTORE

1) the DATA statement. Numerical or string data needed in a program may be written into the program statements themselves, input from peripheral devices or read from DATA statements. The format of the DATA statement is as follows:

```
     DATA<list>
```

where the entries in the list are numerical or string constants separated by commas. In 4K, expressions may also appear in the list. The effect of the statement is to store the list of values in memory in coded form for access by the READ statement. Examples:

```
10 DATA 1,2,-1E3,.04
20 DATA " LOO", MITS
```

Leading and trailing spaces in string values are suppressed unless the string is enclosed by double quotation marks.

2) The READ statement. The data stored by DATA statements is accessed by READ statements which have the following form:

```
READ<list of variables>
```

where the entries in the list are variable names separated by commas. The effect of the READ statement is to assign the values in the DATA lists to the corresponding variables in the READ statement list. This is done one by one from left to right until the READ list is exhausted. If there are more names in the READ list than values in the DATA lists, an OD or OUT OF DATA error message is issued. 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 one DATA statement, and more than one READ statement may access the data in a single DATA statement.

An SN or SYNTAX ERROR message can result from an improperly formatted DATA list. In 4K Altair BASIC, such an error message will refer to the READ statement which attempted to access the incorrect data. In other versions, the line number in the error message will refer to the actual line of the DATA statement in which the error occurred.

3) RESTORE statement. After the RESTORE statement is executed, the next piece of data accessed by a READ statement will be the first entry of the first DATA list in the program. This allows re-READing the data.

d. CSAVEing and CLOADing Arrays (8K cassette, Extended and Disk versions only). Numeric arrays may be saved on cassette or loaded from cassette using CSAVE* and CLOAD* The formats of the statements are:

```
CSAVE*<array name>
```

and

```
     CLOAD*<array name>
```

The array is written out in binary with four octal 210 header bytes to indicate the start of data. These bytes are searched for when CLOADing the array. The number of bytes written is four plus:

```
      8*<number of elements> for a double precision array
      4*<number of elements> for a single precision array
      2*<number of elements> for an integer array
```

When an array is written out or read in, the elements of the array are written out with the leftmost subscript varying most quickly, the next leftmost second, etc:

```
        DIM A(1Ø)
     CSAVE*A
```

writes out A(Ø),A(1),...A(1Ø)

```
        DIM A(1Ø,1Ø)
     CSAVE*A
```

writes out A(Ø,Ø), A(1,Ø)...A(1Ø,Ø),A(1Ø,1)...A(1Ø,1Ø)

Using this fact, it is possible to write out an array as a two dimensional array and read it back in as a single dimensional array, etc.

NOTE

Writing out a double precision array and reading it back in as a single precision or integer array is not recommended. Useless values will undoubtedly be returned.

e. Miscellaneous Input/Output

1) WAIT (not in 4K). The status of input ports can be monitored by the WAIT command which has the following format:

```
      WAIT<I,J>[,<K>]
```

where I is the number of the port being monitored and J and K are integer expressions. The port status is exclusive ORd with K and the result is ANDed with J. Execution is suspended until a non-zero value results. J picks the bits of port I to be tested and execution is suspended until those bits differ from the corresponding bits of K. Execution resumes at the next statement after the WAIT. If K is omitted, it is assumed to be zero. I, J and K must be in the range Ø to 255. Examples:

| | |
|---|---|
| WAIT 2Ø,6 | Execution stops until either bit 1 or bit 2 of port 2Ø are equal to 1. (Bit Ø is least significant bit, 7 is the most significant.) Execution resumes at the next statement. |
| WAIT 1Ø,255,7 | Execution stops until any of the most significant 5 bits of port 1Ø are one or any of the least significant 3 bits are zero. Execution resumes at the next statement. |

2) POKE, PEEK (not in 4K). Data may be entered into memory in binary form with the POKE statement whose format is as follows:

```
POKE <I,J>
```

where I and J are integer expressions. POKE stores the byte J into the location specified by the value of I. In 8K, I must be less than 32768. In Extended and Disk versions, I may be in the range Ø to 65536. J must be in the range Ø to 255. In 8K, data may be POKEd into memory above location 32768 by making I a negative number. In that case, I is computed by subtracting 65536 from the desired address. To POKE data into location 45ØØØ, for example, I is 45ØØØ-65536=-2Ø536. Care must be taken not to POKE data into the storage area occupied by Altair BASIC or the system may be POKEd to death, and BASIC will have to be loaded again.

The complementary function to POKE is PEEK. The format for a PEEK call is as follows:

```
PEEK(<I>)
```

where 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 Ø and 255. A major use of PEEK and POKE is to pass arguments and results to and from machine language subroutines.

3)OUT, INP (not in 4K). The format of the OUT statement is as follows:

```
OUT <I,J>
```

where I and J are integer expressions. OUT sends the byte signified by J to output port I. I and J must be in the range Ø to 255.

The INP function is called as follows:

```
INP(<I>)
```

INP reads a byte from port I where I is an integer expression in the range Ø to 255. Example:

```
2Ø IF INP(J)=16 THEN PRINT "ON"
```

## 3. FUNCTIONS

Altair BASIC allows functions to be referenced in mathematical function notation. The format of a function call is as follows:

```
<name>(<argument>[,<argument>...])
```

where the name is that of a previously defined function and the arguments are one or more expressions, separated by commas. Only one argument is allowed in 4K and 8K. Function calls may be components of expressions, so statements like

```
1Ø LET T=(F*SIN(T))/P  and
2Ø C=SQR(A^2+B^2+2*A*B*COS(T))
```

are legal.

### 3-1. Intrinsic Functions

Altair BASIC provides several frequently used functions which may be called from any program without further definition. A procedure is provided, however, whereby unneeded functions may be deleted to save memory space. See Appendix B. For a list of intrinsic functions, see section 6-3.

### 3-2. User-Defined Functions (not in 4K).

a. The DEF statement. The programmer may define functions which are not included in the list of intrinsic functions by means of the DEF statement. The form of the DEF statement is as follows:

```
DEF<function name>(<variable list>)=<expression>
```

where the function name must be FN followed by a legal variable name and the entries in the variable list are 'dummy' variable names. The dummy variables represent the argument variables or values in the function call. In 8K Altair BASIC, only one argument is allowed for a user-defined function, but in the Extended and Disk versions, any number of arguments is allowed. Any expression may appear on the right side of the equation, but it must be limited to one line. User-defined functions may be of any type in Extended and Disk versions, but user-defined string functions are not allowed in 8K If a type is specified for the function, the value of the expression is forced to that type before it is returned to the calling statement. Examples:

```
10 DEF FNAVE(V,W)=(V+W)/2
11 DEF FNCON$(V$,W$)=RIGHT$(V$+W$,5)  Returns the right
                               most 5 characters of the concat-
                               enation of V$ and W$.
12 DEF FNRAD(DEG)=3.14159/180*DEG  When called with the
                               measure of an angle in degrees,
                               returns the radian equivalent.
```

A function may be redefined by executing another DEF statement with the same name. A DEF statement must be executed before the function it defines may be called.

b. USR. The USR function allows calls to assembly language subroutines. See appendix E.

## 3-3. Errors.

An FC or ILLEGAL FUNCTION CALL error results when an improper call is made to a function. Some places this might occur are the following:

1. a negative array subscript. LET A(-1)=0, for example.

2. an array subscript that is too large (>32767)

3. negative or zero argument for LOG

4. Negative argument for SQR

5. A^B with A negative and B not an integer

6. a call to USR with no address patched for the machine language subroutine.

7. improper arguments to MID$, LEFT$ ,RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, INSTR, STRING$, SPACE$ or ON...GOTO.

b. An attempt to call a user-defined function which has not previously appeared in a DEF statement will cause a UF or UNDEFINED USER FUNCTION error.

c. A TM or TYPE MISMATCH error will occur if a function which expects a string argument is given a numeric value or vice-versa.

## 4. STRINGS

In all Altair BASIC versions except 4K, expressions may either have numeric value or may be strings of characters. Altair BASIC provides a complete complement of statements and functions for manipulating string data. Many of the statements have already been discussed so only their particular application to strings will be treated in this section.

### 4-1. String Data.

A string is a list of alphanumeric characters which may be from Ø to 255 characters in length. Strings may be stated explicitly as constants or referred to symbolically by variables. String constants are delimited by quotation marks at the beginning and end. A string variable name ends with a dollar sign ($). Examples:

| | |
|---|---|
| A$="ABCD" | Sets the variable A$ to the four character string "ABCD" |
| B9$="14A/56" | Sets the variable B9$ to the six character string "14A/56" |
| FOOFOO$="E$" | Sets the variable FOOFOO$ to the two character string "E$" |

Strings input to an INPUT statement need not be surrounded by quotation marks.

String arrays may be dimensioned exactly as any other kind of array 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 OS or OUT OF STRING SPACE error will result. String space is allocated by the CLEAR command which is explained in section 6-2.

## 4-2. String operations.

a. Comparison Operators. The comparison operators for strings are the same as those for numbers:

```
= equal
<> not equal
< less than
> greater than
=<,<= less than or equal to
=>,>= greater than or equal to
```

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

```
A<Z      ASCII A is Ø65, Z is Ø9Ø
1<A      ASCII 1 is Ø49
" A">"A"  Leading and trailing blanks are significant
             in string literals.
```

b. String Expressions. String expressions are composed of string literals, string variables and string function calls connected by the + or concatenation operator. The effect of the catenation operator is to add the string on the right side of the operator to the end of the string on the left. If the result of concatenation is a string more than 255 characters long, an LS or STRING TOO LONG error message will be issued and execution will be terminated.

c. Input/Output. The same statements used for input and output of normal numeric data may be used for string data, as well.

1) INPUT, PRINT. The INPUT and PRINT statements read and write strings on the terminal. Strings need not be enclosed in quotation marks, but if they are not, leading blanks will be ignored and the string will be terminated when the first comma or colon is encountered. Examples:

```
10 INPUT ZOO$,FOO$          Reads two strings
20 INPUT X$                 Reads one string and assigns
                            it to the variable X$.
30 PRINT X$,"HI, THERE"     Prints two strings, including
                            all spaces and punctuation
                            in the second.
```

2) DATA, READ. DATA and READ statements for string data are the same as for numeric data. For format conventions, see the explanation of INPUT and PRINT above.

### 4-3. String Functions.

The format for intrinsic string function calls is the same as that for numeric functions. For the list of string functions, see section 6-3. Special user-defined string functions are allowed in Extended and Disk versions and may be defined by the use of the DEF statement (see section 3-2). String function names must end with a dollar sign.

## 5. EXTENDED VERSIONS.

The Extended and Disk versions of Altair BASIC provide several statements, operators, functions and commands which are not available either in the 4K or 8K versions. For clarity, these features are grouped together in this section. Some modifications to existing 4K and 8K features, such as the IF...THEN...ELSE statement and number typing facilities, have been discussed in conjunction with the other versions. Check the index for references to those features.

### 5-1. Extended Statements

a. ERASE. The ERASE statement eliminates arrays from a program and allows their space in memory to be used for other purposes. The format of the ERASE statement is as follows:

```
ERASE<array variable list>
```

where the entries in the list are valid array variable names separated by commas. ERASE will only operate on arrays and not array elements. If a name appears in the list which is not used in the program, an ILLEGAL FUNCTION CALL error will occur. The arrays deleted in an ERASE statement may be dimensioned again, but the old values are lost. Example:

```
10 DIM A(5,5)     etc.
      .
      .
      .
60 ERASE A
70 DIM A(100)
```

b. LINE INPUT. It is often desirable to input a whole line to a string variable without use of quotation marks and other delimiters. LINE INPUT provides this facility. The format of the LINE INPUT statement is as follows:

```
LINE INPUT ["<prompt string>",];<string variable name>
```

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 to the carriage return is assigned to the string variable. A LINE INPUT may be escaped by typing Control/C. At that point, BASIC returns to command level and prints OK. Execution may be resumed at the LINE INPUT by typing CONT. LINE INPUT destroys the input buffer, so the command may not be edited by Control/A for re-execution.

c. SWAP. The SWAP statement allows the values of two variables to be exchanged. The format is as follows:

```
SWAP <variable,variable>
```

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. If one or both of the variables are non-array variables which have not had values assigned to them, an ILLEGAL FUNCTION CALL error will result. Both variables must be of the same type or a TYPE MISMATCH error will result. Example:

```
10 INPUT F$,L$
20 SWAP F$,L$
30 PRINT F$,L$
RUN
```

```
?FIRST,LAST                  Data input
LAST          FIRST          Computer prints
```

d. TRON, TROFF. As a debugging aid, two statements are provided to trace the execution of program instructions. 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. Example:

```
TRON                     executed in direct mode
OK                       printed by computer
10 PRINT 1:PRINT "A"     typed by programmer
20 STOP
RUN
[10] 1                   line numbers and output printed by
A                        computer.
[20]
BREAK IN 20
```

The NEW command will also turn off the trace flag.

e. IF...THEN...ELSE. See section 2-2.

f. DEFINT, DEFSNG, DEFDBL, DEFSTR. See section 2-1

g. CONSOLE, WIDTH. CONSOLE allows the console terminal to be switched from one I/O port to another. The format of the statement is:

```
CONSOLE <I/O port number>,<switch register setting>
```

The <I/O port number> is the hardware port number of the low order (status) port of the new I/O board. This value must be a numeric expression between 0 and 255 inclusive. If it is not in this range, an ILLEGAL FUNCTION CALL error will occur. The <switch register setting> is also a value between 0 and 255 inclusive which specifies the type of I/O port (SIO, PIO, 4PIO etc) being selected. Appropriate values of the <switch register setting> may be found in Appendix B in the table of sense switch settings or in the table below.

Table of values for <switch register setting>:

| I/O Board | Sense Switch Setting |
|---|---|
| 2SIO with 2 stop bits | Ø |
| 2SIO with 1 stop bit | 1 |
| SIO | 2 |
| ACR | 3 |
| 4PIO | 4 |
| PIO | 5 |
| HSR | 6 |
| non-standard terminal | 14 |
| no terminal | 15 |

## WIDTH Statement

The WIDTH statement sets the width in characters of the printing terminal line. The format of the WIDTH statement is as follows:

```
     WIDTH <integer expression>
```

Example:

```
     WIDTH 8Ø
     WIDTH 32
```

The <numeric formula> must have a value between 15 and 255 inclusive, or an ILLEGAL FUNCTION CALL error will occur.

h. Error Trapping. Extended and Disk Altair BASIC make it possible for the user to write error detection and handling routines which can attempt to recover from errors or provide more complete explanation of the cause of errors than the simple error messages. This facility has been added to Altair BASIC through the use of the ON ERROR GOTO, RESUME and ERROR statements and with the ERR and ERL variables.

1) Enabling Error Trapping. The ON ERROR GOTO statement specifies the line of the Altair BASIC program on which the error handling subroutine starts. The format is as follows:

```
     ON ERROR GOTO <line number>
```

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.

Example:

```
10 ON ERROR GOTO 1000
```

2) Disabling the Error Routine. ON ERROR GOTO 0 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 0 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 0 subroutine if an error is encountered for which they have 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.

3) 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 section 6-5 for a detailed discussion of each of the errors and error messages.

| Code | Error |
|---|---|
| 1 | NEXT WITHOUT FOR |
| 2 | SYNTAX ERROR |
| 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 |

| | |
|---|---|
| 1Ø | 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 | UNPRINTABLE ERROR |
| 20 | NO RESUME |
| 21 | RESUME WITHOUT ERROR |
| 22 | MISSING OPERAND |
| 23 | LINE BUFFER OVERFLOW |

Disk Errors

| | |
|---|---|
| 5Ø | FIELD OVERFLOW |
| 51 | INTERNAL ERROR |
| 52 | BAD FILE NUMBER |
| 53 | FILE NOT FOUND |
| 54 | BAD FILE MODE |
| 55 | FILE ALREADY OPEN |
| 56 | DISK NOT MOUNTED |
| 57 | DISK I/O ERROR |
| 58 | FILE ALREADY EXISTS |
| 59 | SET TO NON-DISK STRING |
| 6Ø | DISK ALREADY MOUNTED |
| 61 | DISK FULL |
| 62 | INPUT PAST END |
| 63 | BAD RECORD NUMBER |
| 64 | BAD FILE NAME |
| 65 | MODE-MISMATCH |
| 66 | DIRECT STATEMENT IN FILE |
| 67 | TOO MANY FILES |
| 68 | OUT OF RANDOM BLOCKS |

The ERL variable contains the line number of the line where the error was detected. For instance, if the error occured in line 1ØØØ, ERL will be equal to 1ØØØ. If the statement which caused the error was a direct mode statement, ERL will 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 (see section 1-1a).

4) Disk Error Values - The ERR function. The ERR function returns the parameters of a DISK I/O ERROR. ERR(Ø) returns the number of the disk, ERR(1) returns the track number (Ø-76) and ERR(2) returns the sector number (Ø-31). ERR(3) and ERR(4) contain the low and high order bytes, respectively, of the cumulative error count since BASIC was loaded.

NOTE

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

5) The RESUME statement. The RESUME statement is used to continue execution of the BASIC program after the error recovery procedure has been performed. The user has three options. The user may RESUME execution at the statement that caused the error, at the statement after the one that caused the error or at some other line. To RESUME execution at the statement which caused the error, the user should use:

```
RESUME
```

or

```
RESUME Ø
```

To RESUME execution at the statement immediately after the one which caused the error, the user should use:

```
RESUME NEXT
```

To RESUME execution at a line dfferent than the one where the error occurred, use:

```
RESUME <line number>
```

Where <line number> is not equal to zero.

6) Error Routine Example. The following example shows how a simple error trapping subroutine operates.

```
100 ON ERROR GOTO 500
200 INPUT "WHAT ARE THE NUMBERS TO DIVIDE";X,Y
210 Z=X/Y
220 PRINT "QUOTIENT IS";Z
230 GOTO 200
500 IF ERR=11 AND ERL=210 THEN 520
510 ON ERROR GOTO 0
520 PRINT "YOU CANT HAVE A DIVISOR OF ZERO!"
530 RESUME 200
```

7) The ERROR statement. In order to force branching to an error trapping routine, an ERROR statement has been provided. The primary use of the ERROR statement is to allow the user to define his own error codes which can then conveniently be handled by a centralized error trap routine as described above. The format of the ERROR statement is:

```
ERROR <integer expression>
```

When defining error codes, values should be picked which are greater than the ones used by Altair BASIC. Since more error messages may be added to Altair BASIC, user-defined error codes should be assigned the highest possible numbers to assure future compatibility. If the <numeric 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 Altair 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.

## 5-2. Extended Operators.

Two operators are provided that are exclusive to the Extended and Disk versions.

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 divison. Integer division is approximately eight times as fast as standard floating point division.

b. Modulus Arithmetic - the MOD operator. A MOD B 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.

## 5-3. Extended Functions

a. Intrinsic Functions. Extended and Disk Altair BASIC provide several intrinsic functions which are not available in the other versions. For a list of these functions and a description of their use, see section 6-3.

b. The DEFUSR statement. Up to ten assembly language subroutines may be defined by means of the DEFUSR statement whose form is as follows:

```
DEFUSR[<digit Ø through 9>]=<integer expression>
```

Example:

```
DEFUSR1=&1ØØØØØ
DEFUSR2=31Ø96
DEFUSR9=ADR
```

The of the <integer expression> is the starting address of the USR routine specified. When the USR subroutine is entered, the A register contains the type of the argument which was given to the USR function. This is also the length of the descriptor for that argument type:

| Value in A | Meaning |
|---|---|
| 2 | Two byte signed two's complement integer. |
| 3 | String. |
| 4 | Single precision four byte floating point number. |
| 8 | Double precision floating point number. |

When the USR subroutine is entered, the [H,L] register pair contains a pointer to the floating point accumulator (FAC). The [H,L] registers contain the address of FAC-3.
If the value in the FAC is a single precision floating point number, it is stored as follows:

| | |
|---|---|
| FAC-3: | Lowest 8 bits of mantissa. |
| FAC-2: | Middle 8 bits of mantissa. |
| FAC-1: | Highest 7 bits of mantissa with hidden (implied) leading one. Bit 7 is the sign of the number (Ø positive, 1 negative). |

FAC: Exponent excess 2ØØ octal. If the contents of FAC is 2ØØ, the exponent is Ø. If contents of FAC is Ø,the number is zero.

If the argument is double precision floating point, the FAC-7 to FAC-4 contain four more bytes of mantissa, low order byte in FAC-7, etc. If the argument is an integer, FAC-3 contains the low order byte and FAC-2 contains the high order byte of the signed two's complement value. If the argument is a string, [D,E] points to a string descriptor of the argument, whose form is:

| Byte | Use |
|---|---|
| Ø | Length of string Ø-255 decimal. |
| 1-2 | Sixteen bit address pointer to first byte of strings text in memory (Caution - may point into program text if argument is a string literal). |

Normally, the value returned by a USR function will be the same type (integer, string, single or double precision floating point) as the argument which was passed to it. However, calling the MAKINT routine whose address is stored in location 6 will return the integer in [H,L] as the value of the function, forcing the value returned by the function to be integer. Execute the following sequence to return from the function:

```
        PUSH    H                   ;SAVE VALUE TO BE RETURNED
        LHLD    6                  ;GET ADDRESS OF MAKINT ROUTINE
        XTHL                        ;SAVE RETURN ON STACK &
                                      ;GET BACK [H,L]
        RET                         ;RETURN
```

The argument of the function may be forced to an integer, no matter what its type by calling the FRCINT routine whose address is located in location 4 to get the integer value of the argument in [H,L]:

```
        LXI     H,SUB1              ;GET ADDRESS OF SUBROUTINE
                                    ;CONTINUATION
        PUSH    H                   ;PLACE ON STACK
        LHLD    4                   ;GET ADDRESS OF FRCINT
        PCHL                        ;CALL FRCINT

SUB1: .....
```

## 5-4. The EDIT Command.

The EDIT command allows modifications and additions to be made to existing program lines without having to retype the entire line each time. Commands typed in the EDIT mode are, as a rule, not echoed. That is, they usually do not appear on the terminal screen or printout as they are typed. Most commands may be preceded by an optional numeric repetition factor which may be used to repeat the command a number of times. This repetition factor should be in the range Ø to 255 (Ø is equivalent to 1). If the repetition factor is omitted, it is assumed to be 1. In the following examples, a lower case "n" before the command stands for the repetition factor. In the following description of the EDIT commands, the "cursor" refers to a pointer which is positioned at a character in the line being edited.

To EDIT a line, type EDIT followed by the number of the line and hit the carriage return. The line number of the line being EDITed will be printed followed by a space. The cursor will now be positioned to the left of the first character in the line.

NOTE

The best way of getting the "feel" of the EDIT command is to try EDITing a few lines yourself.

If a command not recognized as an EDIT command is entered, the computer prints a bell (control/G) and the command is ignored.

In the following examples, the lines labelled "computer prints" show the appearance of the line after each command.

a. Moving the Cursor. Typing a space moves the cursor to the right and causes the character passed over to be printed. A number preceding the space (n<space>) will cause the cursor to pass over and print out n characters. Typing a Rubout causes the immediately previous character to be printed effectively backspacing the cursor.

b. Inserting Characters

WARNINGS:

Character insertion is stopped by typing Escape (or Altmode on some terminals). Control/C will not interrupt the EDIT command while it is in Insert mode, but will be inserted into the edited line. Therefore, Control/C should not be used in the EDIT command.

It is possible using EDIT to create a line which, when listed with its line number, is longer than 72 characters. Punched paper tapes containing such lines will not read properly. However, such lines may be CSAVEd and CLOADed without error.

I Inserts new characters into the line being edited. Each character typed after the I is inserted at the current cursor position and printed on the terminal. Typing Escape (or Altmode on some terminals) stops character insertion. If an attempt is made to insert a character that will make the line longer than 255 characters, a Control/G (bell) is sent to the terminal and the character is not printed.

A backarrow (or Rubout) typed during an insert command (or-) will delete the character to the left of the cursor. Characters up to the beginning of the line may be deleted in this manner, and a backarrow will be echoed for each character deleted. However, if there are no characters to the left of the cursor, a bell is echoed instead of a backarrow. If a carriage return is typed during an insert command, it is as if an escape and then carriage return were typed. That is, all characters to the right of the cursor will be printed and the EDITed line will replace the original line.

X X is similar to I, except that all characters to the right of the cursor are printed, and the cursor moves to the end of the line. At this point, it will automatically enter the insert mode ( see I command). X is most useful when new statements are to be added to the end of an existing line. For example:

```
User types          EDIT 50 (carriage return)
Computer prints     50
User types             X
Computer prints     50 X=X+1
User types                  :Y=Y+1(CR)
Computer prints     50 X=X+1:Y=Y+1
```

In the above example, the original line #50 was:

```
50      X=X+1
```

The new line #50 now reads:

```
50  X=X+1:Y=Y+1
```

H  H is the same as X, except that all characters to the right of the cursor are deleted (they will not be printed). The insert mode (see I command) will then automatically be entered. H is most useful when the last statements on a line are to be replaced with new ones.

c. Deleting Characters

D  nD deletes n characters to the right of the cursor. If n is ommitted, it defaults to 1. If there are less than n characters to the right of the cursor, characters will be deleted only to the end of the line. The cursor is positioned to the right of the last character deleted. The characters deleted are enclosed in backslashes (\). For example:

```
User types         20 X=X+1:REM JUST INCREMENT X
User types         EDIT 20 (carriage return)
Computer prints    20
User types            6D (carriage return)
Computer prints    20 \X=X+1:\REM JUST INCREMENT X
```

The new line #20 will no longer contain the characters which are enclosed by the backslashes.

d. Searching.

S  The nSy command searches for the nth occurrence of the character y in the line. N defaults to 1. The search skips over the first character to the right of the cursor and begins with the second character to the right of the cursor. All characters passed over during the search are printed. If the character is not found, the cursor will be at the end of the line. If it is found, the cursor will stop to the right of the character and all of the characters to its left will have been printed. For example

```
User types  :          50 REM INCREMENT X
User types  :          EDIT 50
```

```
Computer prints        50
User types  :             2SE
Computer prints        50 SUM INCR
```

K nKy is equivalent to S except that all of the characters passed over during the search are deleted. The deleted characters are enclosed in backslashes. For example:

```
User types             10 TEST LINE
User types             EDIT 10
Computer prints        10
User types                KL
Computer prints        10 \TEST \
```

e. Text Replacement.

C A character in a line may be changed by the use of the command Cy which changes the character to the right of the cursor to the character y. Y is printed on the terminal and the cursor is advanced one position. nCy may be used to change n characters in a line as they are typed in from the terminal. (See example below.) If an attempt is made to change a character which does not exist, the change mode will be exited. Example:

```
User types             10 FOR I=1 TO 100
User types             EDIT 10
Computer prints        10
User types                2S1
Computer prints        10 FOR I=1 TO
User types                           3C256
Computer prints        10 FOR I=1 TO 256
```

f. Ending and Restarting

Carriage Return Terminates editing and prints the remainder of the line. The edited line replaces the original line.

E E is the same as a carriage return, except the remainder of the line is not printed.

Q Q restores the original line and causes BASIC to return to command level. Changes do not take effect until an E or carriage return is typed, so Q allows the user to restore the original line without any changes which may have been made.

L L causes the remainder of the line to be printed, and then prints the line number and restarts editing at the beginning of the line. The cursor will be positioned to the left of the first character in the line. L allows monitoring the effect of changes on a line. Example:

```
User types          50 REM INCREMENT X
User types          EDIT 50
Computer prints     50
User types             2SM
Computer prints     50 REM INCRE
User types                   L
Computer prints     50 REM INCREMENT X
                    50
```

A A causes the original line to be restored and editing to be restarted at the beginning of the line. For example:

```
User types          10 TEST LINE
User types          EDIT 10
Computer prints     10
User types             10D
Computer prints     10 \TEST LINE\
User types                      A
Computer prints     10 \TEST LINE\
                    10
```

In the above example, the user made a mistake when he deleted TEST LINE. Suppose that he wants to type "1D" instead of 10D. As a result of the A command, the original line 10 is reentered and is ready for further editing.

IMPORTANT

Whenever a SYNTAX ERROR is discovered during the execution of a source program , BASIC will automatically begin EDITing the line that caused the error as if an EDIT command had been typed. Example:

```
10 APPLE
RUN
SYNTAX ERROR IN 10
10
```

Complete editing of a line causes the line edited to be reinserted. Reinserting a line causes all variable values to be deleted. To preserve those values for examination, the EDIT command mode may be exited with the Q command after the line number is printed. If this is done, BASIC will return to command level and all variable values will be preserved.

The features of the EDIT command may be used on the line currently being typed. To do this, type Control/A instead of Carriage Return. The computer will respond with a carriage return, an exclamation point (!) and a space. The cursor will be positioned at the first character of the line. At this point, any of the EDIT subcommands except Control/A may be used to correct the line. Example:

```
User types          1Ø IF X GOTO #"/A
Computer prints     !
User types            S#            2C12
Computer prints     ! 1Ø IF X GOTO 12
```

The current line number may be designated by a period (.) in any command requiring a line number. Examples:

```
User types          1Ø FOR I= 1 TO 1Ø
User types          EDIT .
Computer prints     1Ø
```

## 5-5. PRINT USING statement.

The PRINT USING statement can be employed in situations where a specific output format is desired. This situation might be encountered in such applications as printing payroll checks or accounting reports. The general format for the PRINT USING statement is as follows:

```
PRINT USING <string>;<value list>
```

The <string> may be a string variable , string expression or a string constant which is a precise copy of the line to be printed. All of the characters in the string will be printed just as they appear, with the exception of the formatting characters. The <value list> is a list of the items to be printed. The string will be repeatedly scanned until: 1) the string ends and there are no values in the value list or, 2) a field is scanned in the string, but the value list is exhausted. The string is constructed according to the following rules:

a. String Fields.

! specifies a single character string field.
(The string itself is specified in the value list.)
\n spaces\ Specifies a string field consisting of 2+n characters. Backslashes with no spaces between them

would indicate a field of 2 characters width, one space between them would indicate a field 3 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:

```
10 A$="ABCDE":B$="FGH"
20 PRINT USING "!";A$;B$
30 PRINT USING "\  \";B$;A$
```

(the above would print out)

```
AF
FGH ABCD
```

Note that where the "!" was used only the first letter of each string was printed. Where the backslashes enclosed two spaces, four letters from each string were printed (an extra space was printed for B$ which has only three characters). The extra characters in the first case and for A$ in the second case were ignored.

b. 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 # sign, each of which will represent a digit position. These digit positions are always filled. The numeric field will be 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 as necessary. If the field format specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). |

The following program will help illustrate these rules:

```
    1Ø INPUT A$,A
    2Ø PRINT USING A$;A
    3Ø GOTO 1Ø
    RUN
    ? ##,12
     12
    ? ###,12
      12
    ? #####,12
        12
    ?##.##,12
     12.ØØ
    ? ###.,12
     12.
    ? #.###,.Ø2
     Ø.Ø2Ø
    ?##.#,2.36
      2.4
?###,-12
    -12
?#.##,-.12
  -.12
?####,-12
   -12
```

| | |
|---|---|
| + | 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. |
| - | The - sign, when used to the right of the numeric field designation, 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. |
| ** | The ** placed at the beginning of a numeric field designation will cause any unused spaces in the leading portion of the number printed out to be filled with asterisks. The ** also specifies positions for 2 more digits. (Termed "asterisk fill") |
| $$ | When the $$ is used at the beginning of a numeric field designation, a $ sign will be printed in the space immediately preceding the number printed. Note that $$ also specifies positions for two more digits, but that the $ itself takes up one of these spaces. Exponential format cannot be used with leading $ signs, nor can negative numbers be output |

unless the sign is forced to be trailing.

**$ The **$ used at the beginning of a numeric field designation causes both of the above (** and $$) to be performed on the number being printed out. All of the previous conditions apply, except that **$ allows for 3 additional digit positions, one of which is the $ sign.

, 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.

^^^^ (↑↑↑↑ on some terminals) Exponential Format. If exponential format is desired in the printout, the numeric field designation should be followed by ^^^^ (allows space for E+XX). Any decimal point arrangement is allowed. The significant digits are left justified and the exponent is adjusted. Unless a leading + or a trailing + or - is used, one position to the left of the decimal point will be used to print a space or minus sign. Examples:

```
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
```

% 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 Altair 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

```
%1.00.
```

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

The following program will help illustrate the preceding rules.

```
Program: 10 INPUT A$,A
         20 PRINT USING A$;A
         30 GOTO 10
         RUN
```

The computer will start by typing a ?. The numeric field designator and value list are entered and the output is displayed as follows:

```
? +#,9
+9
? +#,10
%+10
? ##,-2
-2
? +#,-2
-2
? #,-2
%-2
? +.###,.02
+.020
? ####.#,100
 100.0
? ##+,2
 2+
? THIS IS A NUMBER ##,2
THIS IS A NUMBER  2
? BEFORE ## AFTER,12
BEFORE 12 AFTER
? ####,44444
%44444
? **##,1
***1
? **##,12
**12
? **##,123
*123
? **##,1234
1234
? **##,12345
%12345
? **,1
*1
? **,22
```

```
22
? **.##,12
12.ØØ
? **#####,1
*****1
(note: not floating $)        ? $####.##,12.34
                              $  12.34
(note: floating $)            ? $$####.##,12.56
                               $12.56
                              ? $$.##,1.23
                              $1.23
                              ? $$.##,12.34
                              %$12.34
                              ? $$###,Ø.23
                                 $Ø
                              ? $$####.##,Ø
                                   $Ø.ØØ
                              ? **$###.##,1.23
                              ****$1.23
                              ? **$.##,1.23
                              *$1.23
                              ? **$###,1
                              ****$1
? #,6.9
 7
? #.#,6.99
 7.Ø
? ##-,2
 2
? ##-,-2
 2-
? ##+,2
 2+
? ##+,-2
 2-
? ##^^^^,2
 2E+ØØ
? ##^^^^,12
 1E+Ø1
? #####.###^^^^,2.45678
 2456.78ØE-Ø3
? #.###^^^^,123
 Ø.123E+Ø3
? #.##^^^^,-123
-.12E+Ø3
? "#####,###.#",1234567.89
 1,234,57Ø.Ø
```

Typing Control/C will stop the program.

## 5-6. Disk file operations.

As many as sixteen floppy disks may be connected to a single ALTAIR disk controller. These disks have been assigned the physical disk numbers Ø through 15. Users with one drive should address the drive at zero, and users with two drives should address them at zero and one, etc.

In the following descriptions, <disk number> is an integer expression whose value is the physical number of one of the disks in the system. If the <disk number> is omitted from a statement other than MOUNT or UNLOAD, the <disk number> defaults to Ø. If the <disk number> is omitted from a MOUNT or UNLOAD statement, disks Ø through the highest disk number specified at initialization are affected.

a. Opening, Closing and Naming Files. To initialize disks for reading and writing, the the MOUNT command is issued as follows:

```
MOUNT [<disk number>[,<disk number>...]]
```

Example:

```
MOUNT Ø
```

Mounts the disk on drive zero, and

```
MOUNT Ø,1
```

Mounts the disks on drives zero and one. If there is already a disk MOUNTed on the specified drive(s) a DISK ALREADY MOUNTED message will be printed. Before removing a disk which has been used for reading and writing by Disk Altair BASIC, the user should give an UNLOAD command:

```
UNLOAD [<disk number>[,<disk number>...]]
```

UNLOAD closes all the files open on a disk, and marks the disk as not mounted. Before any further I/O is done on an UNLOADed disk, a MOUNT command must be given.

NOTE

MOUNT, UNLOAD or any other disk command may be used as a program statement.

All data and program files on the disk have an associated file name. This name is the result of evaluating a string expression and must be one to eight characters in length. The first character of the file name cannot be a null (Ø) byte or a byte of 255 decimal. An attempt to use a null file name (zero characters in length) , a file name over 8 characters in length or containing a Ø or 255 in the first character position will cause a BAD FILE NAME error. Any other sequence of one to eight characters is acceptable.

Examples of valid file names:

```
    ABC
    abc              (Not the same as ABC)
    filename
    file.ext
    12345678
    INVNTORY
    FILE##22
```

NOTE

Commands that require a file name will use <file name> in the appropriate position. Remember that a <file name> can be any string expression as long as the resulting string follows the rules given above.

b. The FILES Command. The FILES command is used to print out the names of the files residing on a particular disk. The format of the FILES command is:

```
FILES <disk number>
```

Example:

```
FILES              (prints directory of files on disk Ø)

STRTRK  PIP  CURFIT  CISASM
```

Execution of the FILES command may be interrupted by typing Control/C. A more complete listing of the information stored in a particular file may be obtained by running the PIP utility program (see Appendix I).

c. SAVEing and LOADing programs. Once a program has been written, it is often desirable to save it on a disk for use at a later time. This is accomplished by issuing a SAVE command:

```
SAVE <file name>[,<disk number>[,A]]
```

Example:

```
SAVE "TEST",Ø
```

or

```
SAVE "TEST"
```

would save the program TEST on disk zero. 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. See section 5-6d for a discussion of saving with the 'A' option.

The LOAD statement reads a file from disk and loads it into memory. The syntax of the LOAD statement is:

```
LOAD <file name>[,<disk number>[,R]]
```

Correspondingly:

```
LOAD "TEST",Ø or LOAD "TEST"
```

loads the program TEST from disk zero. If the file does not exist, a FILE NOT FOUND error will occur.

```
LOAD "TEST",Ø,R

OK
```

LOADs the program TEST from disk zero and runs it. The LOAD command with the "R" option may be used to chain or segment programs into small pieces if the whole program is too large to fit in the computer's memory. All variables and program lines are deleted by LOAD, but all data files are kept OPEN(see below) 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 (see below) by a LOAD.

Example:

```
NEW
1Ø PRINT "FOO1":LOAD "FOO2",Ø,R
SAVE "FOO1",Ø

OK
1Ø PRINT "FOO2":LOAD "FOO1",Ø,R
SAVE "FOO2",Ø
```

```
OK
RUN
FOO2
FOO1
FOO2
FOO1
...etc.
```

(Control/C may be used to stop execution at this point)

In this example, program FOO2 is RUN. FOO2 prints the message "FOO2" and then calls the program FOO1 on disk. FOO1 prints "FOO1" and calls the program FOO2 which prints "FOO2" and so on indefinitely.

RUN may also be used with a file name to load and run a program. The format of the command is as follows:

```
RUN<file name>[,<disk number>[,R]]
```

All files are closed unless ,R is specified after the disk number.

d. SAVEing and LOADing Program Files in ASCII. Often it is desirable to save a program in a form that allows the program text to be read as data by another program, such as a text editor or resequencing program. Unless otherwise specified, Altair BASIC saves its programs in a compressed binary format which takes a minimum of disk space and loads very quickly. To save a program in ASCII, specify the "A" option on the SAVE command:

```
SAVE "TEST",Ø,A

OK

LOAD "TEST",Ø

OK
```

Information in the file tells the LOAD command the format in which the file is to be loaded. The first character of an ASCII file is never 255, and a binary program file always starts with 255 (377 octal). Remember, loading an ASCII file is much slower than loading a binary file.

e. The MERGE Command. Sometimes it is very useful to put parts of two programs together to form a new program combining elements of both programs. The MERGE command is provided for this purpose. As soon as the MERGE command has been executed, BASIC returns to command level. Therefore it is more likely that MERGE would be used as a direct command than as a statement in a program. The format of the MERGE statement is as follows:

```
MERGE <file name>[,<disk number>]
```

Example:

```
MERGE "PRINTSUB",1
OK
```

The <file name> specified is merged into the program already in memory. The <file name> must specify an ASCII format saved program 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.

f. Deleting Disk Files. The KILL statement deletes a file from disk and returns disk space used by the file to free disk space. The format of the KILL statement is as follows:

```
KILL <file name>[,<disk number>]
```

If the file does not exist, a FILE NOT FOUND error will occur. If a KILL statement is given for a file that is currently OPEN (see below), a FILE ALREADY OPEN error occurs.

g. Renaming Files - the NAME Statement. The NAME statement is used to change the name of a file:

```
NAME <old file name> AS <new file name>[,<disk number>]
```

Example:

```
NAME "OLDFILE" AS "NEWFILE"
```

The <old file name> must exist, or a FILE NOT FOUND error will occur. A file with the same name as <new file name> 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.

h. OPENing Data Files. Before a program can read or write data to a disk file, it must first OPEN the file on the appropriate disk in one of several modes. The general form of the OPEN statement is:

```
OPEN <mode>,[#]<file number>,<file name>[,<disk number>]
```

<mode> is a string expression whose first character is one of the following:

| | |
|---|---|
| O | Specifies sequential output mode |
| I | Specifies sequential input mode |
| R | Specifies random Input/Output mode |

A sequential file is a stream of characters that is read or written in order much like INPUT and PRINT statements read from and write to the terminal. Random files are divided into groups of 128 characters called records. The nth record of a file may be read or written at any time. Random files have other attributes that will be discussed later in more detail.

<file number> is an integer expression between one and fifteen. The number is associated with the file being OPENed and is used to refer to the file in later I/O operations.

Examples:

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

The above two statements would open the file OUTPUT for sequential output and the file INPUT for sequential input on disk zero.

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

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

i. Sequential ASCII file I/O Sequential input and output files are the simplest form of disk input and output since they involve the use of the INPUT and PRINT statements with a file that has been previously OPENed.

INPUT is used to read data from a disk file as follows:

INPUT #<file number>,<variable list>

where <file number> represents the number of the file that was OPENed for input and <variable list> is a list of the variables to be read, as in a normal INPUT statement. When data is read from a sequential input 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 to a standard INPUT statement to the terminal. When reading numeric values, leading spaces, carriage returns and line feeds are ignored. When a non-space, non-carriage return, non-line-feed character is found, it is assumed to be part of a number in Altair BASIC format. The number terminates on a space, a carriage return , line-feed or a comma.

When scanning for string items, leading blanks, carriage returns and line-feeds are also ignored. When a character which is not a leading blank, carriage return or line-feed 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, carriage return or line feed. 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 seen.

Sequential I/O commands destroy the input buffer so they may not be edited by Control/A for re-execution.

Example of sequential I/O (numeric items):

```
500 OPEN "O",1,"FILE",0
510 PRINT #1,X,Y,Z
520 CLOSE #1
```