altair 8800 basic

REFERENCE MANUAL
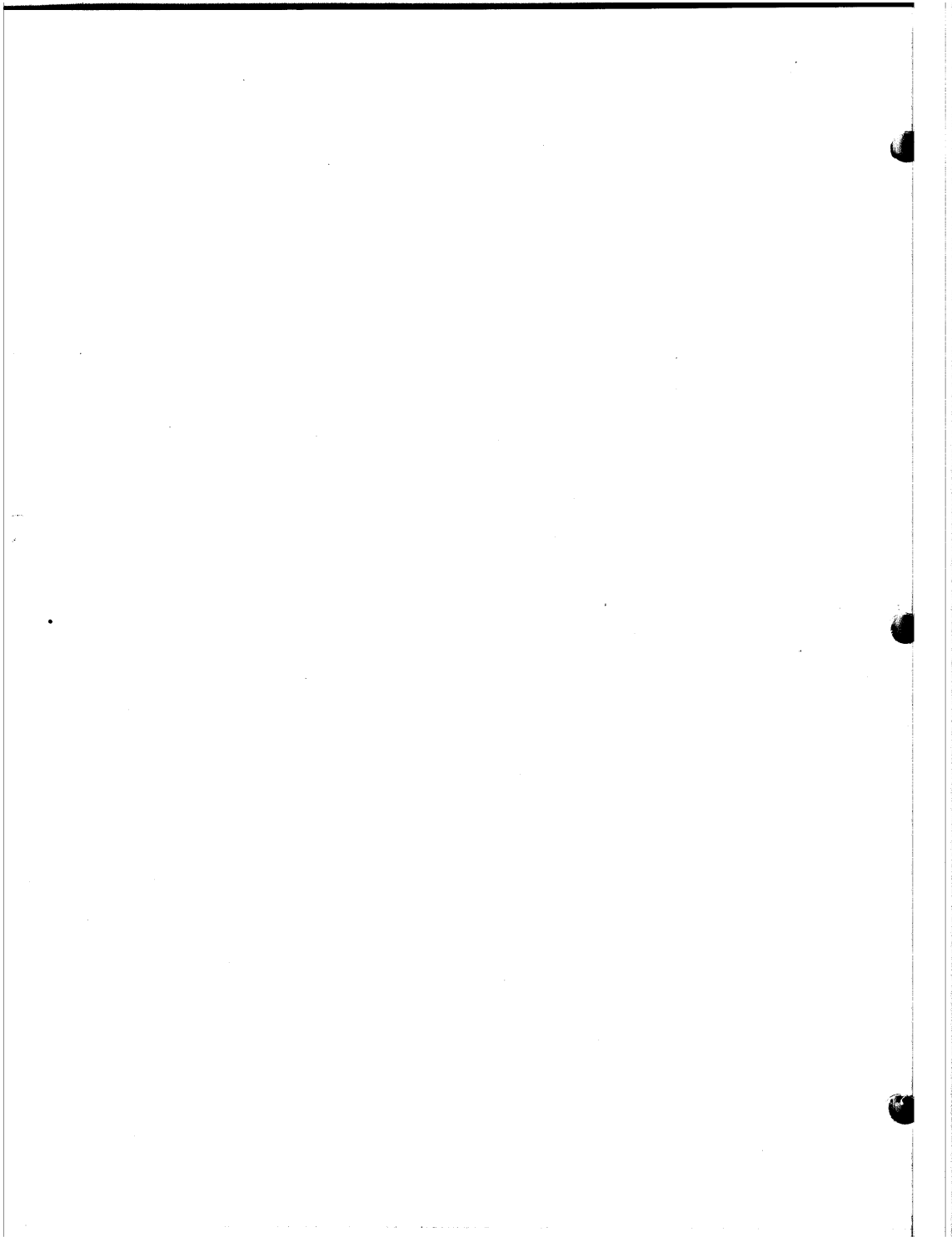
BASIC Reference Manual

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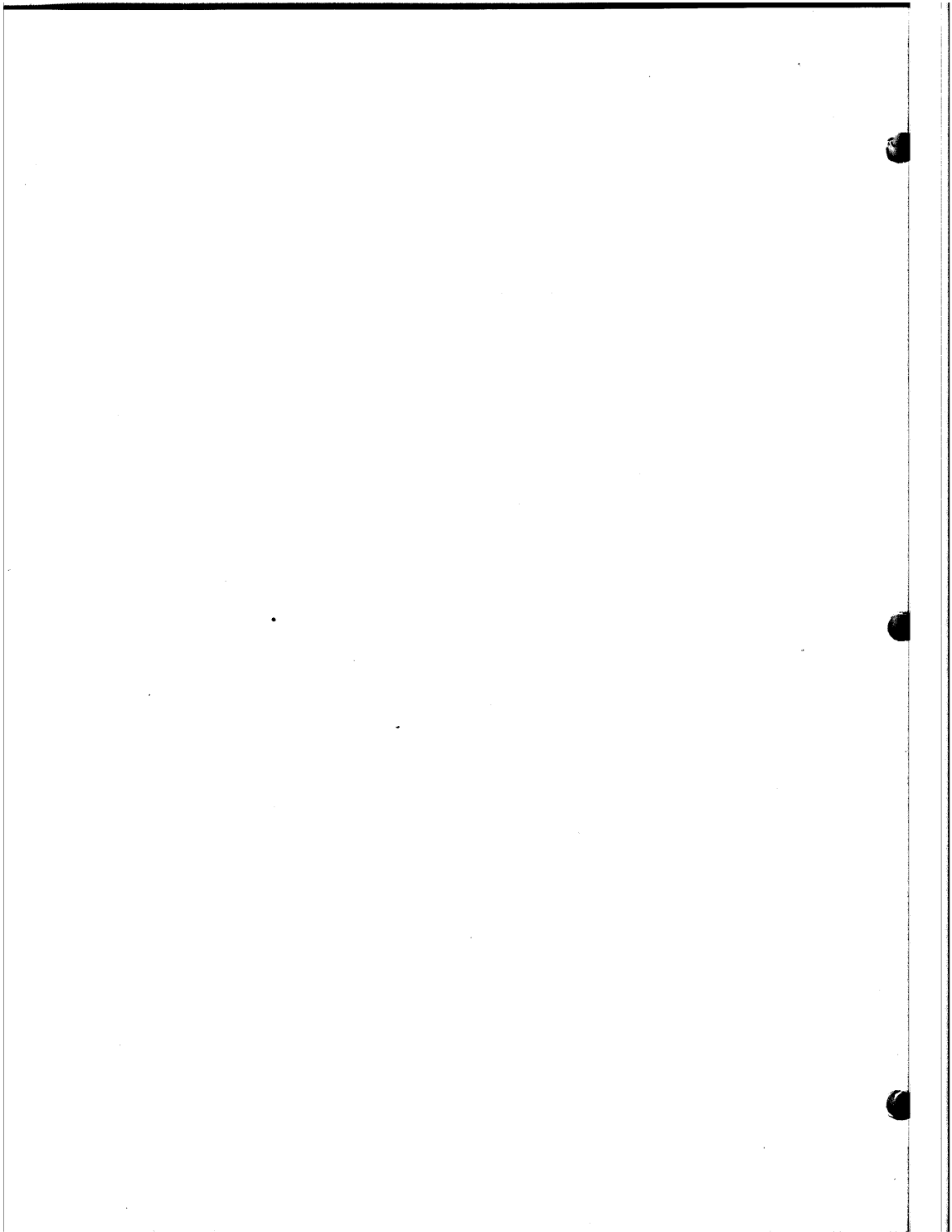
      A string returned by a user's routine should be completely within
   the bounds of the storage area used by the original string. Increasing
   a string's length in a user routine is guaranteed to cause problems.

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.

5. Page 59, last line:

    520 CLOSE #1

    CHANGE TO:

    520 <u>CLOSE 1</u>

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 <u>(see 4-1)</u> to the value . . .

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

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

    CHANGE TO:

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

8. Page 71:

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

    CHANGE TO:

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

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

    ADDITION:

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

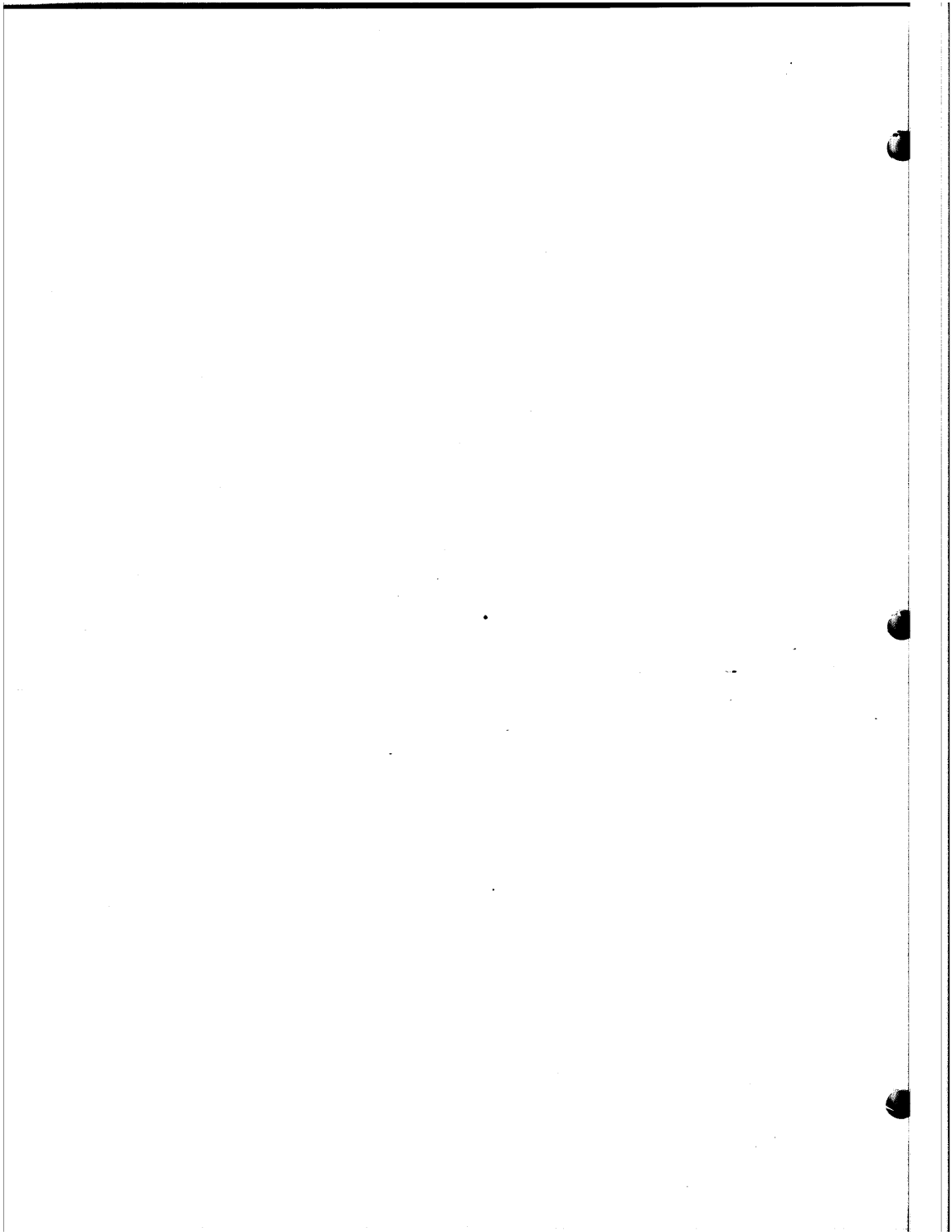   Extended, Disk

    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 <u>65536+I</u>, . . .

11. Page 80, OCT$:

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

CHANGE TO:

OCT$ OCT$(X)   <u>Extended, Disk</u>

12. Page 81:

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

CHANGE TO:

SPACE$ SPACE$(I)   <u>Extended, Disk</u>

13. Page 91, line 4:

. . . question (see Appendix E).

CHANGE TO:

. . . question (see <u>Appendix H)</u>.

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 <u>Appendix H</u>.

15. Page 103, line 11:

C (in extended) retains CONSOLE function.

CHANGE TO:

C (in Extended <u>and Disk</u>) retains CONSOLE <u>and all other functions</u>.

16. Page 112, Paragraph 4, Line 3:

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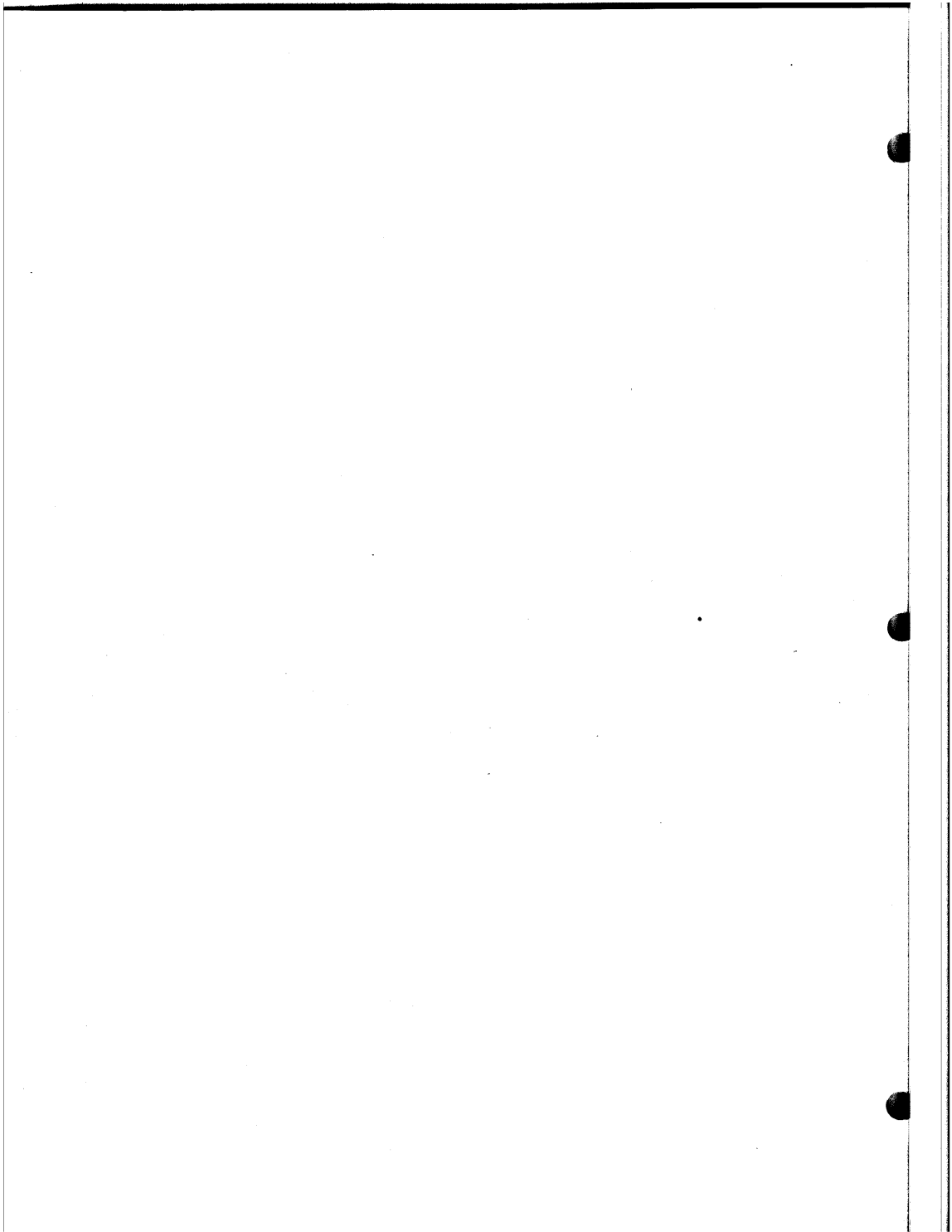
CHANGE TO:

USRLOC for 4K and 8K Altair BASIC version 4.0 is 111 <u>octal</u>.

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. Page 119, last sentence before the NOTE:

File numbers are calculated by multiplying the sector number of the direc-
tory track the file is in by 16 and adding the position of the slot in the
sector (0-8) plus 1.

CHANGE TO:

File numbers are calculated by multiplying the sector number of the direc-
tory track the file is in by 8 and adding the position of the slot in the
sector (0-7) plus 1.

19. Page 122, Step 1, line 3:

. . . location 2=116 octal . . .

CHANGE TO:

. . . location 2=077 octal . . .

20. Page 126, line 6:

COPØ,1 FROM Ø TO 1? YCARRIAGE return> DONE

CHANGE TO:

*COPØ,1
FROM Ø TO 1? Y<carriage return>
DONE
*

21. Page 126, lines 13 through 15:

. . . Example: *DATØ (DAT is equivalent) TRACK? ØSECTOR? Ø ØØØ ØØØ ØØØ
ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ etc.
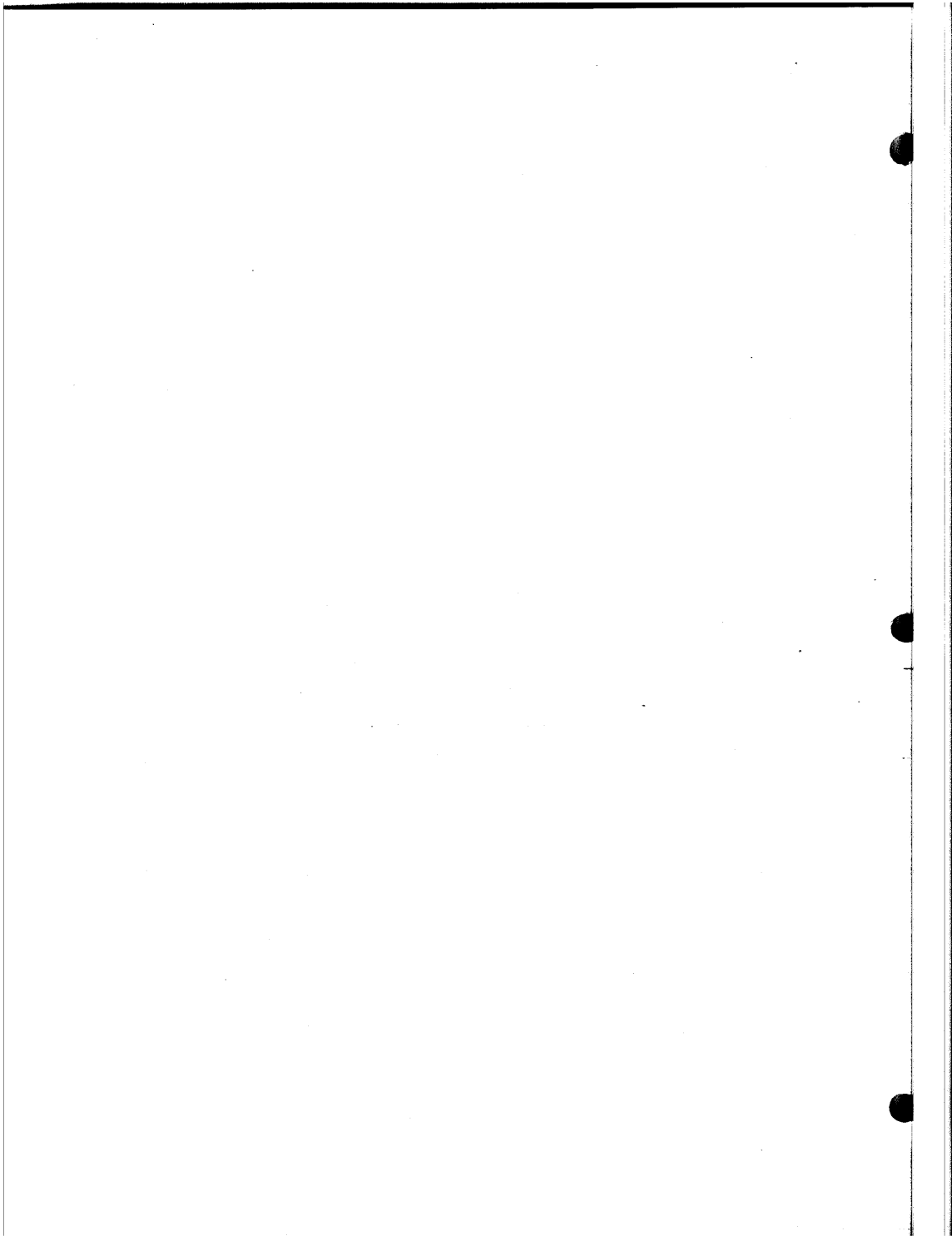
CHANGE TO:

*DATØ
TRACK? Ø
SECTOR? Ø
ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ ØØØ etc.

22. Page 131, line 1 of program:

ORG   7Q1

CHANGE TO:

ORG   71Q

23. Page 135, Step 7, line 2:

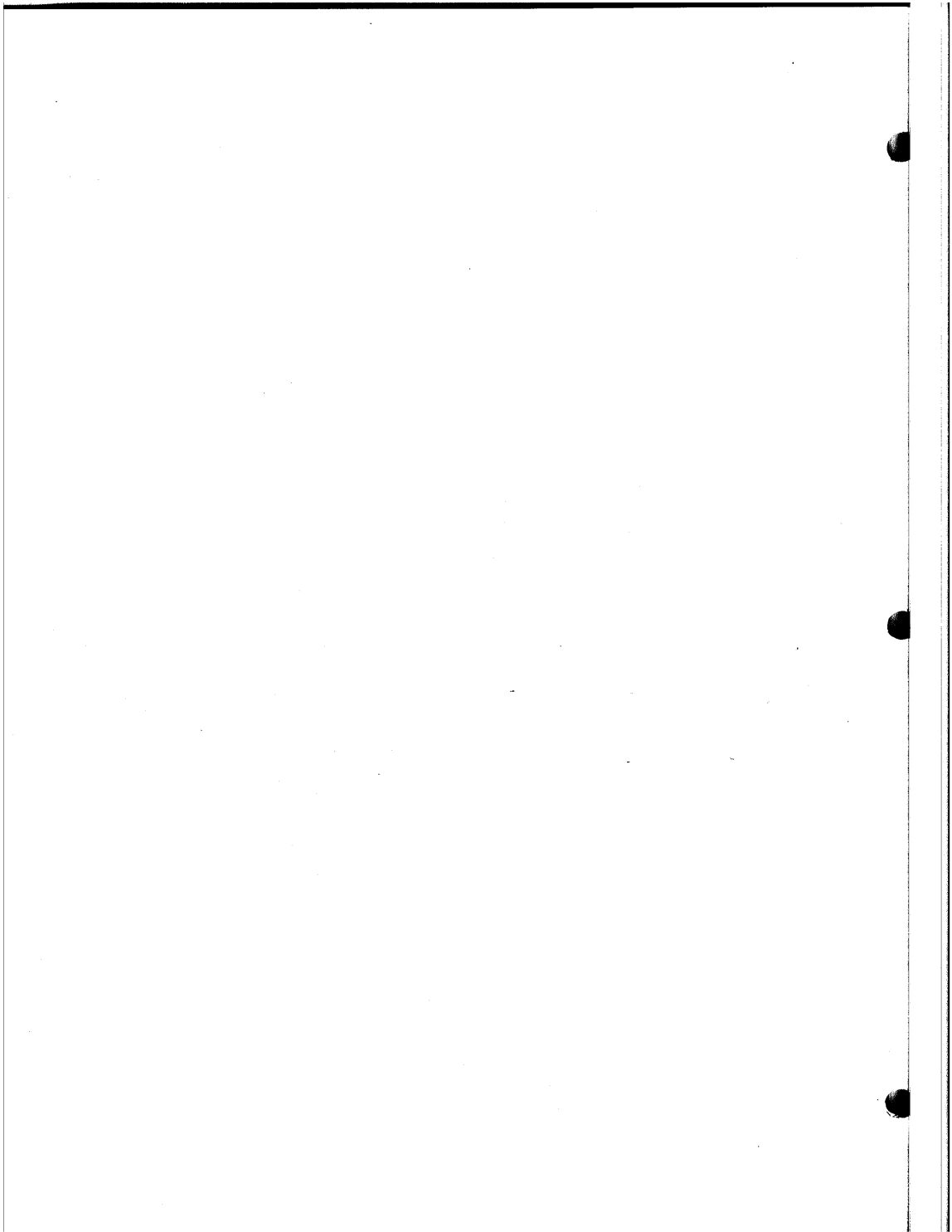   . . . the board type is IOCHNL, . . .

    CHANGE TO:

    . . . the board <u>address</u> is IOCHNL, . . .

24. Index, line 12:

    ADDITION:

    NULL 72

PREFACE

The Altair BASIC language is a high-level programming
language specifically designed for interactive computing
systems. Its simple English-like instructions are easily
understood and quickly learned and its interactive nature
allows instant feedback of results and diagnostics. Despite
its simplicity, however, Altair BASIC has evolved into a
powerful language with provisions for editing and string
processing as well as numerical computation.

The Altair BASIC interpreter reads the instructions of
the BASIC language and directs the ALTAIR 8800 series
microcomputer to execute them. Altair BASIC includes many
useful diagnostic and editing features in all versions. The
extended versions provide additional features including
comprehensive file input/output procedures in the disk
version.

This manual will explain the features of the BASIC
language and the special provisions of the 4K, 8K, Extended
and Disk Extended Altair BASIC interpreters, release 4.0.
For quick reference, a table of Altair BASIC instructions,
diagnostics and functions are provided in Section 6. A
complete index is at the end of the manual. In addition to
this reference manual, the programmer should have a good
BASIC text book on hand. A list of some suggested texts is
given in Appendix J.

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 0 to 65529. A good programming practice is to use an increment of 5 or 10 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 100,10
        100 INPUT X,Y
        110 PRINT SQR(X^2+Y^2)
        120 ^C
        OK
```

AUTO will number every input line until Control/C is typed. If the <initial line> is omitted, it is assumed to be 10 and an increment of 10 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 10. 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 10. Examples:

RENUM   Renumbers the whole program to start at line 10 with an increment of 10 between the new line numbers.

RENUM 100,,100   Renumbers the whole program to start at line 100 with an increment of 100.

RENUM 6000,5000,1000   Renumbers the lines from 5000 up so they start at 6000 with an increment of 1000.

NOTE

RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) 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
    0.0436
    1.25E+05

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 1.234567890123

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 0 to 999999, it is printed as an integer.

3. If the absolute value of the number is greater than or
    equal to .01 and less than or equal to 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
1.70141E38, 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
1E20	1E20
-12.34567E-10	-1.23456E-09
1.234567E-7	1.23457E-07
1000000	1E+06
.1	.1
.01	.01
.000123	1.23E-04
-25.460	-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 0 - 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 can-
                                            not 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 let-
                        ters 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 parenthe-
              ses 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 dy-
                                  namically during program
                                  execution. At the time the
                                  DIM is executed, the expression
                                  within the parentheses is e-
                                  valuated and the results trun-
                                  cated 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.

                        Table of Precedence

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 /0 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

10. 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 (0).

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 0 is the least significant.

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

NOT

X	NOT X
1	0
0	1

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

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

63 AND 16=16	63=binary 111111 and 16=binary 10000, so 63 AND 16=16
15 AND 14=14	15= binary 1111 and 14=binary 1110, so 15 AND 14=binary 1110=14.
-1 AND 8=8	-1=binary 1111111111111111 and 8=binary 1000, so -1 AND 8=8.
4 OR 2=6	4=binary 100 and 2=binary 10 so 4 OR 2=binary 110=6.
10 OR 10=10	binary 1010 OR'd with itself is 1010= 10.
-1 OR -2=-1	-1=binary 1111111111111111 and -2= 1111111111111110, so -1 OR -2=-1.
NOT 0=-1	the bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	the two's complement of any number is the bit complement plus one.

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:

1000 LET V=X
110 LET I=I+1   the '=' sign heremeans 'is replaced
                   by ....'

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

120 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
1.70141E38 or -1E-38 to -1.70141E38. An attempt to divide
by zero results in the /0 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:

```
10 IF A=10 THEN 40      If the expression A=10 is
   true, BASIC branches  to  line  40.    Otherwise,
   execution proceeds at the next line.
15 IF A<B+C OR X THEN 100  The expression after IF is
   evaluated and if the value  of  the  expression  is
   non-zero,  the  statement  branches  to  line  100.
```

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=0 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:

      10 FOR I=2 TO 11   The loop is executed 10 times with
                           the variable I taking on each in-
                           tegral value from 2 to 11.
      20 FOR V=1 TO 9.3   This loop will execute 9 times un-
                           til V is greater than 9.3
      30 FOR V=10*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.
      40 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(10)
    CSAVE*A

writes out A(0),A(1),...A(10)

        DIM A(10,10)
    CSAVE*A

writes out A(0,0), A(1,0)...A(10,0),A(10,1)...A(10,10)

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 0 to 255. Examples:

    WAIT 20,6   Execution stops until either bit 1 or bit
                  2 of port 20 are equal to 1. (Bit 0 is
                  least significant bit, 7 is the most sig-
                  nificant.) Execution resumes at the next
                  statement.

    WAIT 10,255,7 Execution stops until any of the most significant
                  5 bits of port 10 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 0 to 65536. J must be in the range 0 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 45000, for example, I is
45000-65536=-20536. 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 0
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 0 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 0 to 255. Example:

        20 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

        10 LET T=(F*SIN(T))/P and
        20 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 charac-
                    ter 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 065, Z is 090
    1<A   ASCII 1 is 049
    " 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	0
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 80
        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

```
10     REDIMENSIONED ARRAY
11     DIVISION BY ZERO
12     ILLEGAL DIRECT
13     TYPE MISMATCH
14     OUT OF STRING SPACE
15     STRING TOO LONG
16     STRING FORMULA TOO COMPLEX
17     CAN'T CONTINUE
18     UNDEFINED USER FUNCTION
19     UNPRINTABLE ERROR
20     NO RESUME
21     RESUME WITHOUT ERROR
22     MISSING OPERAND
23     LINE BUFFER OVERFLOW
```

Disk Errors

```
50     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
60     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 1000, ERL will be equal to 1000. 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(0) returns the number of the disk, ERR(1) returns the track number (0-76) and ERR(2) returns the sector number (0-31). 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 0

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=0, 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 0 through 9>]=<integer expression>

Example:

DEFUSR1=&100000
DEFUSR2=31096
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 (0
        positive, 1 negative).

FAC: Exponent excess 200 octal. If the contents of FAC is 200,
       the exponent is 0. If contents of FAC is 0,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
0   Length of string 0-255 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 0 to 255 (0 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.

<div align="center">NOTE</div>

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

<div align="center">WARNINGS:</div>

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 REM 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 re-
         mainder 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          10 IF X GOTO #"/A
Computer prints     !
User types           S#            2C12
Computer prints     ! 10 IF X GOTO 12
```

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

```
User types          10 FOR I= 1 TO 10
User types          EDIT .
Computer prints     10
```

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, out 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 char-
        acters. 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:

```
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
RUN
? ##,12
  12
? ###,12
   12
? #####,12
     12
?##.##,12
  12.00
? ###.,12
   12.
? #.###,.02
  0.020
?##.#,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
                    12.00
                    ? **####,1
                    *****1
                    (note: not floating $)    ? $####.##,12.34
                                              $   12.34
                    (note: floating $)        ? $$####.##,12.56
                                               $12.56
                                              ? $$.##,1.23
                                              $1.23
                                              ? $$.##,12.34
                                              %$12.34
                                              ? $$###,0.23
                                                $0
                                              ? $$####.##,0
                                                $0.00
                                              ? **$###.##,1.23
                                              ****$1.23
                                              ? **$.##,1.23
                                              *$1.23
                                              ? **$###,1
                                              ****$1
                    ? #,6.9
                     7
                    ? #.#,6.99
                     7.0
                    ? ##-,2
                     2
                    ? ##-,-2
                     2-
                    ? ##+,2
                     2+
                    ? ##+,-2
                     2-
                    ? ##^^^^,2
                     2E+00
                    ? ##^^^^,12
                     1E+01
                    ? #####.###^^^^,2.45678
                     2456.780E-03
                    ? #.###^^^^,123
                     0.123E+03
                    ? #.##^^^^,-123
                    -.12E+03
                    ? "#####,###.#",1234567.89
                     1,234,570.0
```

Typing Control/C will stop the program.

5-6.   <u>Disk</u> <u>file</u> <u>operations.</u>

As many as sixteen floppy disks may be connected to a single ALTAIR disk controller. These disks have been assigned the physical disk numbers 0 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 0. If the <disk number> is omitted from a MOUNT or UNLOAD statement, disks 0 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 0

Mounts the disk on drive zero, and

MOUNT 0,1

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",0

  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",0 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",0,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
        10 PRINT "FOO1":LOAD "FOO2",0,R
        SAVE "FOO1",0

        OK
        10 PRINT "FOO2":LOAD "FOO1",0,R
        SAVE "FOO2",0
```

```
        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",0,A

        OK

        LOAD "TEST",0

        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",0
    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
```

```
530 OPEN "I",1,"FILE",0
540 INPUT #1",X,Y,Z
```

Note that CLOSE is used so that a file which has just been
written may be read. When FILE is re-OPENed, the data
pointer for that file is set back to the beginning of the
file so that the first INPUT on the file will read data from
the start of the file.

2) PRINT and PRINT USING statements are used to write
data into a sequential output file. Their formats are as
follows:

    PRINT #<file number>,<expression list>

or

    PRINT #<file number>,
     USING <string expression>;<expression list>

    Example of sequential I/O (quoted string items):

```
500 OPEN "O",1,"FILE"
510 PRINT #1,CHR$(34);X$;CHR$(34);
515 PRINT #1,CHR$(34);Y$;CHR$(34);CHR$(34);Z$;CHR$(34)
520 CLOSE 1
530 OPEN "I",1,"FILE",0
540 INPUT #1,X$,Y$,Z$
```

In this example, the strings being output (X$, Y$, Z$) are
surrounded with double quotes through the use of the CHR$
function to generate the ASCII value for a double quote.
This technique must be used if a string which is being
output to a sequential data file contains commas, carriage
returns, line-feeds or leading blanks that are significant.
When leading blanks are not significant and there are no
commas, carriage returns or line-feeds in the strings to be
output, it is sufficient to insert commas between the
strings being output as in the following example:

```
500 OPEN "O",1,"FILE"
510 PRINT #1,X$;",";Y$;",";Z$
520 CLOSE 1
530 OPEN "I",1,'FILE",0
540 INPUT #1,X$,Y$,Z$
```

3) CLOSE. The format of the CLOSE statement is as
follows:

    CLOSE [<file number>[,<file number>...]]

CLOSE is used to finish I/O to a particular Altair BASIC data file. After CLOSE has been executed for a file, the file may be reOPENed for input or output on the same or different <file number>. A CLOSE for a sequential output file writes the final buffer of output. A CLOSE to any OPEN file finishes the connection between the <file number> and the <file name> given in the OPEN for that file. It allows the <file number> to be used again in another OPEN statement.

A CLOSE with no argument CLOSEs all OPEN files.

NOTE

A FILE can be OPENed for sequential input or random access on more than one <file number> at a time but may be OPEN for output on only one <file number> at a time.

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

4) LINE INPUT. Often it is desirable to read a whole line of a file into a string without using quotes, commas or other characters as delimiters. This is especially true if certain fields of each line are being used to contain data items, or if a BASIC program saved in ASCII mode is being read as data by another program. The facility provided to perform this function is the LINE INPUT statement:

LINE INPUT #<file number>,<string variable>

A LINE INPUT from a data file will return all characters up to a carriage return in <string variable>. LINE INPUT then skips over the following carriage return/line-feed sequence so that a subsequent LINE INPUT from the file will return the next line.

5) End of File (EOF) Detection. When reading a sequential data file with INPUT statements it is usually desirable to detect when there is no more data in the disk file. The mechanism for detecting this condition is the EOF function:

X=EOF(<file number>)

EOF returns TRUE (-1) when there is no more data in the file and FALSE (Ø) otherwise. If an attempt is made to INPUT

past the end of a data file, an INPUT PAST END error will
occur.

    Example:

    100 OPEN "I",1,"DATA",0
    110 I=0
    120 IF EOF(1) THEN 160
    130 INPUT #1,A(I)
    140 I=I+1
    150 GOTO 120
    160 ......

In this example, numeric data from the sequential input file
DATA is read into the array A. When end of file is
detected, the IF statement at line 120 branches to line 160,
and the variable I "points" one beyond the last element of A
that was INPUT from the file.

    The following is a program that will calculate the
number of lines in a BASIC program file that has been SAVEd
in ASCII mode:

    10 INPUT "WHAT IS THE NAME OF THE PROGRAM";P$
    20 OPEN "I",1,P$,0
    30 I=0
    40 IF EOF(1) THEN 70
    50 I=I+1:LINE INPUT #1,L$
    60 GOTO 40
    70 PRINT "PROGRAM ";P$;" IS ";I;" LINES LONG"
    80 END

This example uses the LINE INPUT statement to read each line
of the program into the "dummy" string L$ which is used just
to INPUT and ignore that part of the file.

    6) Finding the Amount of Free Disk Space (DSKF). It is
sometimes necessary to determine the amount of free disk
space remaining on a particular disk before allocating
(writing) a file. The DSKF function provides the user with
the number of free groups left on a given disk, after the
disk has been MOUNTed. A group is the fundamental unit of
file allocation. That is, files are always allocated in
groups of eight sectors at a time. Each sector contains 128
characters (bytes). Therefore, the minimum size for a file
is 1024 bytes.

Syntax for the DSKF function:

    DSKF(<disk number>)

    Example:

        PRINT DSKF(0)
        200

The above example shows that there are 200*1024=204800
characters (bytes) that can still be stored on disk zero.

        j. RANDOM FILE I/O. Previously, we have discussed how
data may be PRINTed or INPUT from sequential data files.
However, it is often desirable to access data in a random
fashion, for instance to retrieve information on a
particular part number or customer from a large data base
stored on a floppy disk. If sequential files were used, the
whole file would have to be scanned from the start until the
particular item was found. Random files remove this
restriction and allow a program to access any record from
the first to the last in a speedy fashion. Also, random
files transfer data from variables to the disk ouput records
and vice versa in a much faster, more efficient fashion than
sequential files. Random file I/O is more complex than
sequential I/O, and it is recommended that beginners try
sequential I/O first.

        1) OPENing a FILE for Random I/O. Random I/O files are
OPENed just like sequential files.

        OPEN "R",1,"RANDOM",0

When a file is OPENed for random I/O, it is always OPEN for
both input and output simultaneously.

        2) CLOSING Random Files. Like sequential files, random
files must be closed when I/O operations are finished. To
CLOSE a random file, use the CLOSE command as described
previously.

        CLOSE <file number>[,<file number>...]

        3) Reading and writing data to a random file - GET and
PUT. Each random file has associated with it a "random
buffer" of 128 bytes. When a GET or PUT operation is
performed, data is transferred directly from the buffer to
the data file or from the data file to the buffer. The
syntax of GET and PUT is as follows:

PUT [#]<file number>[,<record number>]

GET [#]<file number>[,<record number>]

If <record number> is omitted from a GET or PUT statement, the record number that is one higher than the previous GET or PUT is read into the random buffer. Initially a GET or PUT without a record number will read or write the first record. The largest possible record number is 2046. If an attempt is made to GET a record which has never been PUT, all zeroes are read into the record, and no error occurs.

    4) LOC and LOF. LOC is used to determine what the current record number is for random files. In other words, it returns the record number that will be used if a GET or PUT is executed with the <record number> parameter omitted.

LOC(<file number>)

PRINT LOC(1)
15

LOC is also valid for sequential files, and gives the number of sectors (128 byte blocks) read or written since the OPEN statement was executed.

LOF is used to determine the last record number written to a random file:

LOF(<file number>)

PRINT LOF(2)
200

An attempt to use LOF on a sequential file will cause a BAD FILE MODE error.

The value returned by LOF is always 5 MOD 8. That is , when the value LOF returns is divided by 8, the remainder is always 5. Therefore,the values returned by LOF are 5, 13, 21, 29 etc. This is due to the way random files are allocated.

NOTE

> It is important to note that the value returned by
> LOF may be a record that has never been written in
> by a user program.   This is because of the way
> random files are pre-extended.

     5) Moving Data In and Out of the Random Buffer. So far
we have described techniques for writing (PUT) and reading
(GET) data from a file into its associated random buffer.
Now we will describe how data from string variables is moved
to and from the random buffer itself. This is accomplished
through the use of the FIELD, LSET and RSET statements.

     6) FIELD. The FIELD statement associates some or all
of a file's random buffer with a particular string variable.
Then, when the file buffer is read with GET or written with
PUT, string variables which have been FIELDed into the
buffer will automatically have their contents read or
written. The format of the FIELD statement is:

FIELD [#] <file number> ,<field size> AS <string variable>[...]

<file number> is used to specify the file number of the file
whose random buffer is being referenced. If the file is not
a random file, a BAD FILE MODE error will occur. <field
size> sets the length of the string in the random buffer.
<string variable> is the string variable which is associated
with a certain number of characters (bytes) in the buffer.
Multiple fields may be associated with string variables in a
given FIELD statement. Each successive string variable is
assigned a successive field in the random buffer. Example:

       FIELD 10 AS A$, 20 AS B$, 30 AS C$

The statement above would assign the first 10 characters of
the random buffer to the string variable A$, the next 20
characters to B$ and the next 30 characters to the variable
C$. It is important to note that the FIELD statement does
not cause any data to be transferred to or from the random
buffer. It only causes the string variables given as
arguments to "point" into the random buffer.

     Often, it is necessary to divide the random buffer into
a number of sub-records to make more efficient use of disk
space. For instance, it might be desirable to divide the
128 character record into two identical subrecords. To
accomplish this a "dummy variable" would be placed in the
FIELD statement to represent one of the subrecords. One of
the following statements would be executed depending on
whether the first or second subrecord were needed:

```
        FIELD #1,64 AS D$, 20 AS NAME$,
                20 AS ADDRESSE$, 24 AS OCCUPATION$
```

or

```
    - FIELD #1,20 AS NAME$, 20 AS ADDRESSE$,
                24 AS OCCUPATION$, 64 AS D$
```

where the dummy variable D$ is used to skip over one of the subrecords.   Another way to do the same thing would be to set a variable I that would select the first or second subrecord.

```
        FIELD #1,64*(I-1) AS D$,
          20 AS NAME$, 20 AS ADDRESS$, 24 AS OCCUPATION$
```

Here, if the variable I is one, I-1 *64 =0 characters will be skipped over, selecting the first subrecord. If I is two, 64 characters will be skipped over, selecting the second subrecord. Another technique that is very useful is to use a FOR...NEXT loop and an array to set up subrecords in the random buffer:

```
        1000 FOR I=1 TO 16
        1010 FIELD #1, (I-1)*8 AS D$, 4 AS A$(I),
            4 AS B$(I)                    .
        1020 NEXT I
```

In this example, we have divided the random buffer into 16 subrecords composed of two fields each. The first 4-character field is in A$(X) and the second 4-character field is in B$(X,) where X is the subrecord number.

                                NOTE

    The FIELD statement may be executed any number of
    times on a given file. It does not cause any
    allocation of string space. The only space
    allocation that occurs is for the string variables
    mentioned in the FIELD statement. These string
    variables have a one byte count and two byte pointer
    set up which points into the random buffer for the
    specified file.

7) Using Numeric Values in Random Files: MKI$,   MKS$, MKD$ and CVI, CVS, CVD. As we have seen, data is always stored in the random buffer through the use of string variables.   In order to convert between strings and numbers and vice versa, a number of special functions have been provided.

To convert between numbers and strings:

MKI$(<integer value>)   Returns a two byte string
                                 (FC error if value is not
                                 >=-32768 and <=+32767.
                                 Fractional part is lost)
MKS$(<single precision value>)   Returns a four byte string
MKD$(<double precision value>)   Returns an eight byte string

To convert between strings and numbers:

CVI(<two byte string>)   Returns an integer value
CVS(<four byte string>)   Returns a single precision valu
CVD(<eight byte string>)   Returns a double precision value

CVI, CVS, and CVD all give an ILLEGAL FUNCTION CALL error if the string given as the argument is shorter than required. If the string argument is longer than necessary, the extra characters are ignored.   These functions are extremely fast, since   they   convert   between   Altair   BASIC's   internal representations of integers, single and double precision values   and   strings.   Conventional   sequential   I/O   must perform time-consuming character scanning algorithms when converting between numbers and strings.

8.   LSET and RSET. When a GET operation is performed, all string variables which have been FIELDed into the random buffer for that file automatically have values assigned to them.   The CVI, CVS and CVD functions may be used to convert any numeric fields in the record to their numeric values. When going the other way, i.e. inserting strings into the random buffer before performing a PUT statement, a problem arises.   This is because of the way string assignments usually take place. For example:

LET A$=B$

When a LET statement is executed, B$ is copied into string space, A$ is pointed to the new string and the string length of A$ is modified. However, for assignments into the random buffers we do not want this to happen. Instead, we want the string being assigned to be stored where the string variable was FIELDed.   In order to do this, two special assignment

statements have been provided, LSET and RSET:

    LSET <string variable>=<string expression>

    RSET <string variable>=<string expression>

Examples:
```
    LSET A$=MKS$(V)
    RSET B$="TEST"
    LSET C$(I)=MKD$(D#)
```

The difference between LSET and RSET concerns what happens if the string value being assigned is shorter than the length specified for the string variable in the FIELD statement.   LSET   left   justifies the string, adding blanks (octal 40, decimal 32) to pad out the right side of the string if it is too short. RSET right justifies the string, padding on the left. If the string value is too long, the extra characters at the end of the string are ignored.

                              NOTE

    Do not use LSET or RSET on string variables which
    have not been mentioned in a FIELD statement, or a
    SET TO NON DISK STRING error will occur.

    k. The DSKI$ and DSKO$ Primitives.   Often it is necessary for the user to perform disk I/O operations directly without using any of the normal file structure features of Altair BASIC. To allow this, two special functions have been provided. These are the DSKI$ function and the DSKO$ statement. First we will give examples of how to perform simple disk I/O commands using Altair BASIC statements,

To Enable disk 0:

    OUT 8,0

To Enable disk N:

    OUT 8,N

TO step the disk head out one track:

    WAIT 8,2,2:OUT 9,2

To step the disk head in one track:

    WAIT 8,2,2:OUT 9,1

To test for track 0:

    IF (INP(8) AND 64)=0 THEN <statements or line number>

The above will execute the statements or branch to the line number if the head is positioned at track 0.   This is the outermost track on the disk.

To read sector Y (Y may be any expression, minimum sector =0, maximum = 31):

    A$=DSKI$(Y)

The statement
    DSKO$ <string expression>,<sector expression>

writes the string expression on the sector specified.   The high order bit (most signifigant) of the first character output will always be set to one when the string is written on the sector, and thus will always be one when the sector is read back in using DSKI$. A maximum of 137 characters are written;   giving a string whose length exceeds 137 characters will cause an ILLEGAL FUNCTION CALL error.   If the string argument is less than 137 characters in length, the end of the string will be padded with zeros to make a string of length 137.

6. LISTS AND DIRECTORIES

6-1. Commands.

Commands direct Altair BASIC to arrange memory and
input/output facilities, to list and edit programs and to
handle other housekeeping details in support of program
execution. Altair BASIC accepts commands after it prints
'OK' and is at command level. The table below lists the
commands in alphabetical order. The notation to the right
of the command name indicates the versions to which it
applies.

Command   Version(s)

CLEAR   All

Sets all program variables to zero.

CLEAR[<expression>]   8K, Extended, Disk

Same as CLEAR but sets string space to the value of the
expression.   If no argument is given, string space will
remain unchanged. When Altair BASIC is loaded, string space
is set to 50 bytes in 8K and 200 bytes in extended.

CLOAD<string expression>   8K(cassette), Extended, Disk

Causes the program on cassette tape designated by the first
character of STRING expression> to be loaded into memory. A
NEW command is issued before the program is loaded.

CLOAD?<string expression>   8K(cassette), Extended, Disk

Compares the program in memory with the file on cassette
with the same name. If they are the same, BASIC prints OK.
If not, BASIC prints NO GOOD.

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

Loads the specified array from cassette tape. May be used
as a program statement

CONT   8K, Extended, Disk

Continues program execution after a Control/C has been typed
or a STOP or END statement has been executed. Execution
resumes at the statement after the break occurred unless
input from the terminal was interrupted. In that case,

execution resumes with the reprinting of the prompt (? or prompt string). CONT is useful in debugging, especially where an 'infinite loop' is suspected. An infinite loop is a series of statements from which there is no escape. Typing Control/C causes a break in execution and puts BASIC in command level. Direct mode statements can then be used to print intermediate values, change the values of variables, etc. Execution can be restarted by typing the CONT command, or by executing a direct mode GOTO statement, which causes execution to resume at the specified line number.

     In 4K and 8K Altair BASIC, execution cannot be continued if a direct mode error has occured during the break. In all versions, execution cannot continue if the program was modified during the break.

CSAVE<string expression>   8K(cassette), Extended, Disk

Causes the program currently in memory to be saved on cassette tape under the name specified by the first character of <string expression>.

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

Causes the array named to be saved on cassette tape. May be used as a program statement.

DELETE<line number>   Extended, Disk

Deletes the line in the current program with the specified number. If no such line exists, an ILLEGAL FUNCTION CALL error occurs.

DELETE-<line number>   Extended, Disk

Deletes every line of the current program up to and including the specified line. If there is no such line, an ILLEGAL FUNCTION CALL error occurs.

DELETE<line number>-<line number> Extended, Disk

Deletes all lines of the current program from the first line number to the second inclusive. ILLEGAL FUNCTION CALL occurs if no line has the second number.

EDIT<line number>   Extended, Disk

Allows editing of the line specified without affecting any other lines. The EDIT command has a powerful set of sub-commands which are discussed in detail in section 5-4.

LIST   All

Lists the program currently in memory starting with the
lowest numbered line. Listing is terminated either by the
end of the program or by typing Control/C.

LIST[<line number>]   All

In 4K and 8K, prints the current program beginning at the
specified line. In Extended and Disk, prints the specified
line if it exists.

LIST[<line number>][-<line number>]   Extended, Disk

Allows several listing options.

1.   If the second number is omitted, lists all lines with
     numbers greater than or equal to the number specified.

2.   If the first number is omitted, lists all lines from
     the beginning of the program to the specified line,
     inclusive.

3.   If both line numbers are used, lists all lines from the
     first number to the second, inclusive.

LLIST[<line number>][-<line number>]   Extended, Disk

Same as list with the same options, except prints on the
line printer.

NEW   All

Deletes the current program and clears all variables.   Used
before entering a new program.

NULL<integer expression>   8K, Extended, Disk

Sets the number of nulls to be printed at the end of each
line.   For 10 character per second tape punches, <integer
expression> should be >=3.   For 30 cps punches, it should be
>=3.   When tapes are not being punched, <integer expression>
should be 0 or 1 for Teletypes* and Teletype compatible
CRT's.   It should be 2 or 3 for 30 cps hard copy printers.
The default value is 0.   In the 4K version, the same affect
may be achieved by patching location 46 octal to contain the
number of nulls plus 1.

     * Teletype is a registered trademark of the Teletype
Corporation.

RUN[<line number>]   All

Starts execution of the program currently in memory at the
line specified.   If the line number is omitted, execution
begins at the lowest line number. Line number specification
is not allowed in 4K.

6-2. Statements.

The following table of statements is listed in alpahabetical
order.   The notation in the Version column designates the
versions to which each statement applies. In the table, X
and Y stand for any expressions allowed in the version under
consideration. I and J stand for expressions whose values
are truncated to integers. V and W are any variable names.
The format for a Altair BASIC line is as follows:

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

where nnnnn is the line number.

Name   Format   Version

CONSOLE   CONSOLE <I>,<J>   Extended, Disk

Allows terminal console device to be switched. I is the I/O
port number which is the address of the low order channel of
the new I/O board. J is the switch register setting (see
section 5-1 for the list of settings). 0<=I,J<=255.

DATA   DATA<list>   All

Specifies data to be read by a READ statement.   List
elements can be numbers or, except in 4K, strings. 4K
allows expressions. List elements are separated by commas.

DEF   DEF FNV(<W>)=<X>   8K, Extended, Disk

Defines a user-defined function.   Function name is FN
followed by a legal variable name.   Extended and Disk
versions allow user-defined string functions.   Definitions
are restricted to one line (72 characters in 4K and 8K, 255
characters in extended versions).

DEFUSR   DEFUSR[<digit>]=<X>   Extended, Disk

Defines starting address of assembly language subroutine.
Up to ten subroutines are allowed.

DIM   DIM <V>(<I>[,J...])[,...] All

Allocates space for array variables. In 4K, only one
dimension is allowed per variable. More than one variable
may be dimensioned by one DIM statement up to the limit of
the line.   The value of each expression gives the maximum
subscript possible. The smallest subscript is 0. Without a
DIM statement, an array is assumed to have maximum subscript
of 10 for each dimension referenced. For example, A(I,J) is
assumed to have 121 elements, from A(0,0) to A(10,10) unless
otherwise dimensioned in a DIM statement.

END   END   All

Terminates execution of a program. Closes all files in the
Disk version.

ERASE   ERASE<V>[,<W>...]   Extended, Disk

Eliminates the arrays specified. The arrays may be
redimensioned or the space made available for other uses.

ERROR   ERROR<I>   Extended, Disk

Forces error with code specified by the expression. Used
primarily for user-defined error codes.

FOR   FOR<V>=<X>TO<Y>[STEP<Z>] All

Allows repeated execution of the same statements. First
execution sets V=X. Execution proceeds normally until NEXT
is encountered. Z is added to V, then, IF Z<0 and V>=Y, or
if Z>0 and V<=Y, BASIC branches back to the statement after
FOR. Otherwise, execution continues with the statement
after NEXT.

GOTO   GOTO<nnnnn>   All

Unconditional branch to line number

GOSUB   GOSUB<nnnnn>   All

Unconditional branch to subroutine beginning at line nnnnn.

IF...GOTO   IF <X> GOTO<nnnnn>   8K, Extended, Disk

Same as IF...THEN except GOTO can only be followed by a line
number and not another statement.

IF...THEN [ELSE]   IF<X>THEN<X>[ELSE<Y>]   All
                or IF<X>THEN<statement>[:statement...]
                          [ELSE<statement>[:statement...]

If value of X<>0, branches to line number or statement after
THEN.   Otherwise,   branches   to   the   line   number   or
statement(s) after ELSE. If ELSE is omitted, and the value
of X=0, execution proceeds at the line after the IF...THEN.
In 4K, X can only be a numeric expression. The ELSE clause
is only allowed in Extended and Disk Altair BASIC.

INPUT   INPUT<V>[,<W>...]   All

Causes BASIC to request input from terminal. Values (or, in
4K, expressions) typed on the terminal are assigned to the
variables in the list.

LET   LET <V>=<X>   All

Assigns the value of the expression to the variable.   The
word LET is optional.

LPRINT   LPRINT X[,Y...]   Extended, Disk

Same as PRINT, but prints on the line printer.   Line feeds
within strings are ignored. A carriage return is printed
automatically after the 80th character on a line.

LPRINT USING   LPRINT USING<string>;<list> Extended, Disk

Same as PRINT USING, but prints on the line printer. For a
detailed description, see section 5-5.

MID$ MID$(<X$>,<I>[,<J>])=Y$ Extended, Disk

Part of the string X$ is replaced by Y$. Replacement starts
with the Ith character of X$ and proceeds until Y$ is
exhausted, the end of X$ is reached or J characters have
been replaced, whichever comes first. If I is greater than
LEN(X$), an ILLEGAL FUNCTION CALL error results.

NEXT   NEXT [<V>,<W>...]   All

Last statement of a FOR loop. V is the variable of the most
recent loop, W of the next most recent and so on. Only one
variable is allowed in 4K. Except in 4K, NEXT without a
variable terminates the most recent FOR loop.

ON ERROR GOTO   ON ERROR GOTO<line number> Extended, Disk

When an error occurs, branches to line specified.   Sets
variable ERR to error code and ERL to line number where the

error occured. See section 6-5 for a list of error codes.
ON ERROR GOTO Ø (or without number) disables error trapping.

ON...GOTO   ON<I>GOTO<list of line numbers> 8K, Ext., Disk

Branches to line whose number is Ith in the list.   List
elements are separated by commas. If I=Ø or > number of
elements in the list, execution continues at next statement.
If I<Ø or >255, an error results.

ON...GOSUB   ON <I> GOSUB <list>   8K, Extended, Disk

Same as ON...GOTO except list elements are initial line
numbers of subroutines.

OUT   OUT<I>,<J>   8K, Extended, Disk

Sends byte J to port I. Ø<=I,J<=255.

POKE   POKE<I>,<J>   8K, Extended, Disk

Stores byte J in memory location derived from   I.
Ø<=J<=255;-32768<I<65536. If I is negative, address is
65535+I, if I is positive, address=I.

PRINT   PRINT<X>[,<Y>...]   All

Causes values of expressions in the list to be printed on
the terminal. Spacing is determined by punctuation.

Punctuation   Spacing - next printing begins:
     ,   at beginning of next 14 column zone
     ;   immediately
   other or none   at beginning of next line

String literals may be printed if enclosed by (") marks.
String expressions may be printed in all but 4K.

PRINT USING   PRINT USING<string>;<list>   Extended, Disk

Prints the values of the expressions in the list edited
according to the string. The string is an expression which
represents the line to be printed. The list contains the
constants, variable names or expressions to be printed.
List entries are separated by punctuation as in the PRINT
statement. For a list of string characters and their
functions, see section 5-5.

READ   READ<V>[,<W>...]   All

Assigns values in DATA statements to variables. Values are
assigned in sequence starting with the first value in the

first DATA statement.

REM   REM[<remark>]   All

Allows insertion of remarks. Not executed, but may be
branched into. In extended versions, remarks may be added
to the end of a line preceded by a single quotation mark
(').

RESTORE   RESTORE   All

Allows data from DATA statements to be reread. Next READ
statement after RESTORE begins with first data of first data
statement.

RESUME   RESUME[<number>]   Extended, Disk

Resumes program execution at the line specified after error
trapping routine. If number is omitted or zero, resumes at
statement where error occured. RESUME NEXT causes
resumption at the statement following the statement where
the error was made.

RETURN   RETURN   All

Terminates a subroutine. Branches to the statement after
the most recent GOSUB.

STOP   STOP   All

Stops program execution. BASIC enters command level and,
except in 4K, prints BREAK IN LINE nnnnn. Unlike END, STOP
does not close files.

SWAP   SWAP <V>,<W>   Extended, Disk

Exchanges values of the variables named. Variables must be
of the same type.

TROFF   TROFF   Extended, Disk

Turns off trace flag. The trace flag is turned on by TRON
(see below). NEW also turns off the trace flag.

TRON   TRON   Extended, Disk

Turns on trace flag. Prints number of each line in square
brackets as it is executed.

WAIT   WAIT<I>,<J>[,<K>]   8K, Extended, Disk

Status of port I is XOR'd with K and AND'ed with J.

Continued   execution   awaits   non-zero result.   K defaults to
0.   0<=I,J,K<=255.

6-3.   Intrinsic Functions.

        Altair BASIC provides several commonly used algebraic
and string functions which may be called from any program
without   further   definition.   If the functions are not
required for a program, they may be deleted when BASIC is
loaded to conserve memory space.   The functions in the
following table are listed in alphabetical order.   The
notation to the right of the Call Format is the versions   in
which the function is available. As usual, X and Y stand
for expressions, I and J for integer expressions and X$ and
Y$ for string expressions.

Function   Call Format   Version

ABS   ABS(X)   All

Returns absolute value of expression X.   ABS(X)=X if X>=0,
-X if X<0.

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

Returns the ASCII code of the first character of the string
X$.   ASCII codes are in appendix A.

ATN   ATN(X)   8K, Extended, Disk

Returns arctangent(X).   Result is in radians in range -pi/2
to pi/2.

The following functions are available in Extended and Disk:

CINT   CINT(X)   Converts X to integer.
CSNG   CSNG(X)   Converts X to single precision.
CDBL   CDBL(X)   Converts X to double precision.

If the argument is in the range -32768 to 32767, the
CINT(X)=INT(X).   Otherwise,   CINT will produce an OVERFLOW
error.

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

Returns a string whose one element has ASCII code I.   ASCII

codes are in Appendix A.

COS   COS(X)   8K, Extended, Disk

Returns cos(X). X is in radians.

ERL   Extended, Disk

Returns the number of the line in which the last error occurred.

ERR   Extended, Disk

Returns the error code of the last error.

ERR   ERR(I)   Disk

Returns parameters of disk errors. After a DISK I/O ERROR, ERR(0) returns number of the disk, ERR(1) returns the track number (0-76) , ERR(2) returns the sector number, ERR(3) and ERR(4) return the low and high order 8 bits of the cumulative count of disk errors respectively.

EXP   EXP(X)   8K, Extended, Disk

Returns e to the power X. X must be <=87.3365.

FIX   FIX(X)   Extended, Disk

Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

FRE   FRE(0)   8K, Extended, Disk

Returns number of bytes in memory not being used by BASIC. If argument is a string, returns number of free bytes in string space.

HEX$ HEX$(X)   Extended, Disk

Returns a string which represents the hexadecimal of the decimal argument.

INP   INP(I)   8K, Extended, Disk

Reads a byte from port I.

INSTR   INSTR([I,]X$,Y$)   Extended, Disk

Searches for the first occurrence of string Y$ in X$ and

returns the position. Optional offset I sets position for starting the search. 0<=I<=255. If I>LEN(X$), if X$ is null or if Y$ cannot be found, INSTR returns 0. If Y$ is null INSTR returns I or 1. Strings may be string variable values, string expressions or string literals.

INT   INT(X)   All

Returns the largest integer <=X

LEFT$ LEFT$(X$,I)   8K, Extended, Disk

Returns leftmost I characters of string X$.

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

Returns length of string X$. Non-printing characters and blanks are counted.

LOG   LOG(X)   8K, Extended, Disk

Returns natural log of X. X>0

LPOS   LPOS(X)   Extended, Disk

Returns the current position of the line printer print head within the line printer buffer. Does not necessarily give the physical position of the print head. The expression X must be given, but the value is ignored.

MID$ MID$(X$,I[,J])   8K, Extended, Disk

Without J, returns rightmost characters from X$ beginning with the Ith character. If I>LEN(X$), MID$ returns the null string. 0<I<255. With 3 arguments, returns a string of length J of characters from X$ beginning with the Ith character. If J is greater than the number of characters in X$ to the right of I, MID$ returns the rest of the string. 0<=J<=255.

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

Returns a string which represents the octal value of the decimal argument.

RND   RND(X)   All

Returns a random number between 0 and 1. X<0 starts a new sequence of random numbers. X>0 gives the next random number in the sequence. X=0 gives the last number returned. In 8K, Extended and Disk, sequences started with the same negative number will be the same.

POS   POS(I)   8K, Extended, Disk

Returns present column position of terminal's print head.
Leftmost position =0.

RIGHT$ RIGHT$(X$,I)   8K, Extended, Disk

Returns rightmost I characters of string X$. If I=LEN(X$),
returns X$.

SGN   SGN(X)   All

If X>0, returns 1, if X=0 returns 0, if X<0, returns -1.
For example, ON SGN(X)+2 GOTO 100,200,300 branches to 100
if X is negative, 200 if X is 0 and 300 if X is positive.

SIN   SIN(X)   All

Returns the sine of the value of X in radians.
COS(X)=SIN(X+3.14159/2).

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

Returns a string of spaces of length I.

SPC   SPC(I)   8K, Extended, Disk

Prints I blanks on terminal. 0<=I<=255.

SQR   SQR(X)   All

Returns square root of X. X must be >=0

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

Returns string representation of value of X.

STRING$ STRING$(I,J)   Extended, Disk

Returns a string of length I whose characters all have ASCII
code J. See Appendix A for ASCII codes.

TAB   TAB(I)   All

Spaces to position I on the terminal. Space 0 is the
leftmost space, 71 the rightmost. If the carriage is
already beyond space I, TAB has no effect. 0<=I<=255. May
only be used in PRINT and LPRINT statements.

TAN   .   TAN(X)   All

Returns tangent(X). X is in radians.

USR   USR(X)   All

Calls the user's machine language subroutine with argument
X.

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

Returns numerical value of string X$. If first character of
X$ is not +,-,& or a digit, VAL(X$)=0.

VARPTR   VARPTR(V)   Extended, Disk

Returns the address of the variable given as the argument.
If the variable has not been assigned a value during the
execution of the program, an ILLEGAL FUNCTION CALL error
will occur.   The main use of the VARPTR function is to
obtain the address of variable or array so it may be passed
to an assembly language subroutine.   Arrays are usually
passed by specifying VARPTR(A[0]) so that the lowest
addressed element of the array is returned.

                             NOTE

    All simple variables should be assigned values in a
    program   before   calling VARPTR for any array.
    Otherwise, allocation of a new simple variable will
    cause the addresses of all arrays to change.

6-4. Special Characters

     Altair BASIC recognizes several characters in the ASCII
font as having special functions in carriage control,
editing and program interruption.   Characters such   as
Control/C, Control/S, etc.   are typed by holding down the
Control key and typing the designated letter.   The special
characters in the table are listed in the order of the
versions to which they apply, starting with those common to
all versions and ending with those that apply only to
extended versions.

Typed as   Printed as

The following Special Characters are available in ALL
versions.

@   @

Erases current line and executes carriage return.

(backarrow)

Erases last character typed. If there is no last character
types a carriage return.

                    _(underline)
—

same as backarrow.

Carriage Return

Returns print head or curser to beginning of the next line.

Control/C   ^C (in extended)

Interrupts execution of current program or list command.
Takes effect after execution of the current statement or
after listing the current line. BASIC goes to command level
and types OK. CONT command resumes execution. See section
6-1.

:   :

Separates statements in a line.

The following special characters are available in 8K,
Extended and Disk versions only.

Control/O   ^O (in extended)

Suppresses all output until   an   INPUT   statement   is
encountered, another Control/O is typed, an error occurs or
BASIC returns to command level.

?   ?

equivalent to PRINT statement.

Rubout   see explanation

Deletes previous character on an input line.   First Rubout
prints \ and the last character to be printed. Each
successive Rubout prints the next character to the left.
Typing a new character causes another \ and the new
character to be printed.   All characters between   the
backslashes are deleted.

Control/U   ^U (in extended)

Same as @

Control/S

Causes program execution to pause until Control/Q or
Control/C is typed.

Control/Q

Causes execution to resume after Control/S. Control/S and
Control/Q have no effect if no program is being executed.

The following special characters are available in Extended
and Disk versions only.

Control/A

Allows use of the EDIT command on the line currently being
typed. Control/A is typed instead of Carriage Return. See
section 5-4.

Control/I   1 to 8 spaces

Tab character. Causes print head or curser to move to the
beginning of the next 8 column field. Fields begin at
columns 1, 9, 17, etc. The tab character is especially
useful for formatting lines broken with line feeds.

```
100<tab>FOR I=1 TO 10:<line feed>
<tab><tab>FOR J=1 TO 10:<line feed>
<tab><tab><tab>A(I,J)=0:<line feed>
<tab>NEXT J,I<carriage return>
```

lists as:

```
100     FOR I=1 TO 10:
                FOR J=1 TO 10:
                        A(I,J)=0:
        NEXT J,I
```

Control/G   bell

Rings terminal's bell

LINE FEED

Breaks a long line into shorter parts. The result is still
one BASIC line.

.

Denotes the number of the current line.   May be used
wherever a line number is to be specified.

[,]   [,]

Brackets are interchangable with parentheses as delimiters
for array subscripts.

Lower Case Input

Lower case alphabetic characters are always echoed as lower
case, but LIST, LLIST, PRINT and LPRINT will translate lower
case to upper case if the lower case characters are not part
of string literals, REM statements or single quote (')
remarks.

6-5.   Error Messages.

After an error occurs, BASIC returns to command level and
types OK.   Variable values and the program text remain
intact, but the program cannot be continued by the CONT
command.   In 4K and 8K versions, all GOSUB and FOR context
is lost. The program may be continued by direct mode GOTO,
however.   When an error occurs in a direct statement, no
line number is printed. Format of error messages:

          Direct Statement   ?XX ERROR
          Indirect Statement   ?XX ERROR IN YYYYY

where XX is the error code and YYYYY is the line number
where the error occurred. The following are the possible
error codes and their meanings:

   ERROR CODE   EXTENDED ERROR MESSAGE   NUMBER

   The following error codes apply in ALL versions.

      BS   SUBSCRIPT OUT OF RANGE   9

An attempt was made to reference an array element which is
outside the dimensions of the array. In the 8K and larger
versions, this error can occur if the wrong number of
dimensions are used in an array reference. For example:

      LET A(1,1,1)=Z

when A has already been dimensioned by DIM A(10,10)

    DD   REDIMENSIONED ARRAY   10

After an array was dimensioned, another dimension statement
for the same array was encountered. This error often occurs
if an array has been given the default dimension of 10 and
later in the program a DIM statement is found for the same
array.

    FC   ILLEGAL FUNCTION CALL   5

The parameter passed to a math or string function was out of
range. FC errors can occur due to:

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

2. an unreasonably large array subscript (>32767)

3. LOG with negative or zero argument

4. SQR with negative argument

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

6. a call to USR before the address of a machine language
    subroutine has been entered.

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

    ID   ILLEGAL DIRECT   12

INPUT and DEF are illegal in the direct mode. In extended
versions, however, INPUT is legal in direct.

    NF   NEXT WITHOUT FOR   1

The variable in a NEXT statement corresponds to no
previously executed FOR statement.

    OD   OUT OF DATA   4

A READ statement was executed but all of the DATA statements
in the program have already been read. The program tried to
read too much data or insufficient data was included in the
program.

OM   OUT OF MEMORY   7

Program is too large, has too many variables, too many FOR
loops, to many GOSUBs or too complicated expressions. See
Appendix C.

OV   OVERFLOW   6

The result of a calculation was too large to be represented
in Altair BASIC's number format. If an underflow occurs,
zero is given as the result and execution continues without
any error message being printed.

SN   SYNTAX ERROR   .   3

Missing parenthesis in an expression, illegal character in a
line, incorrect punctuation, etc.

RG   RETURN WITHOUT GOSUB   3

A RETURN statement was encountered before a previous GOSUB
statement was executed.

UL   UNDEFINED LINE   8

The line reference in a GOTO, GOSUB, IF...THEN...ELSE or
DELETE was to a line which does not exist.

/0   DIVISION BY ZERO   .   11

Can occur with integer division and MOD as well as floating
point division.   0 to a negative power also causes a
DIVISION BY ZERO error.

          The following error messages apply to
           8K, Extended and Disk versions only

CN   CAN'T CONTINUE   17

Attempt to continue a program when none exists, an error
occured, or after a modification was made to the program.

LS   STRING TOO LONG   15

An attempt was made to create a string more than 255
characters long.

OS   OUT OF STRING SPACE   14

String variables exceed amount of string space allocated for

them.   Use the CLEAR command to allocate more string space
or use smaller strings or fewer string variables.

          ST   STRING FORMULA TOO COMPLEX   16

A string expression was too long or too complex.   Break it
into two or more shorter ones.

          TM   TYPE MISMATCH   13

The left hand side of an assignment statement was a numeric
variable   and   the   right   hand   side   was   a string, or
vice-versa;   or a function which expected a string argument
was given a numeric one or vice-versa.

          UF   UNDEFINED USER FUNCTION   18

Reference was made to a user defined function which had
never been defined.

     The following error messages are available in
          Extended and Disk versions only.

                            MISSING OPERAND   22

During evaluation of an expression, an operator was   found
with no operand following it.

                            NO RESUME   20

BASIC entered an error trapping routine, but the program
ended before a RESUME statement was encountered.

                            RESUME WITHOUT ERROR   21

A RESUME statement was encountered, but no   error trapping
routine had been entered.

                            UNPRINTABLE ERROR   19

An error condition exists for which there is no error
message available.   Probably there   is an ERROR statement
with an undefined error code.

                            LINE BUFFER OVERFLOW   23

An attempt was made to input a program or data line which
has too many characters to be held in the line buffer.
Shorten the line or divide it into two or more parts.

Disk Altair BASIC Error Messages

FIELD OVERFLOW   50

An attempt was made to allocate more than 128 characters of
string variables in a single FIELD statement.

INTERNAL ERROR   51

Internal error in Disk BASIC. Report conditions under which
error occurred and all relevant data to MITS software
department.   This error can also be caused by certain kinds
of disk I/O errors.

BAD FILE NUMBER   52

An attempt was made to use a file number which specifies a
file that is not OPEN or that is greater than the number of
files entered during the Disk Altair BASIC initialization
dialog.

FILE NOT FOUND   53

Reference was made in a LOAD, KILL or OPEN statement to a
file which did not exist on the disk specified.

BAD FILE MODE   54,

An attempt was made to perform a PRINT to a random file, to
OPEN a random file for sequential output, to perform a PUT
or GET on a sequential file, to load a random file or to
execute an OPEN statement where the file mode is not I, O,
or R.

FILE ALREADY OPEN   55

A sequential output mode OPEN for a file was issued for a
file that was already OPEN and had never been CLOSEd or a
KILL statement was given for an OPEN file.

DISK NOT MOUNTED   56

An I/O operation was issued for a file that was not MOUNTed.

DISK I/O ERROR   57

An I/O error occured on disk X. A sector read (checksum)
error occurred eighteen (18) consecutive times.

SET TO NON-DISK STRING   58

An LSET or RSET was given for a string variable which had
not previously been mentioned in a FIELD statement.

DISK ALREADY MOUNTED   59

A MOUNT was issued for a DISK that was already MOUNTed but
never UNLOADed.

DISK FULL   60

All disk storage is exhausted on the disk. Delete some old
disk files and try again.

INPUT PAST END   61

An INPUT statement was executed after all the data in a file
had been INPUT. This will happen immediately if an INPUT is
executed for a null (empty) file. Use of the EOF function
to detect End Of File will avoid this error.

BAD RECORD NUMBER   62

In a PUT or GET statement, the record number is either
greater than the allowable maximum (2046) or equal to zero.

BAD FILE NAME   63

A file name of 0 characters (null) or a file name whose
first byte was 0 or 377 octal (255 decimal) or a file name
with more than 8 characters was used as an argument to LOAD,
SAVE, KILL or OPEN.

MODE-MISMATCH   64

Sequential OPEN for output was executed for a file that
already existed on the disk as a random (R) mode file, or
vice versa.

DIRECT STATEMENT IN FILE   65

A direct statement was encountered during a LOAD of a
program in ASCII format. The LOAD is terminated.

TOO MANY FILES   66

A SAVE or OPEN (O or R) was executed which would create a
new file on the disk, but all 255 directory entries were
already full. Delete some files and try again.

OUT OF RANDOM BLOCKS   67

An attempt was made to have more random files OPEN at once
than the number of random blocks that were allocated during
initialization   by   the   response   to   the
"NUMBER OF RANDOM FILES?" question (see Appendix E).

FILE ALREADY EXISTS   68

The new file name specified in a NAME statement had the same
name as another file that already existed on the disk.   Try
a different name.

FILE LINK ERROR   69

During the reading of a file, a sector was read which did
not belong to the file.

6-6. Reserved Words.

Some words are reserved by the Altair BASIC interpreter for
use as statements, commands, operators, etc. and thus may
not be used for variable or function names.   The reserved
words are listed below in order of the versions for which
they are reserved, starting with those reserved in all
versions and ending with those reserved only in Disk Altair
BASIC. Words reserved in larger versions may be used in
smaller versions, although one may want to avoid all
reserved words in the interest of compatibility.   In
addition to the words listed below, intrinsic function names
are reserved words in all versions in which they are
available.

RESERVED WORDS

Words reserved in all versions.
CLEAR   NEW
DATA   NEXT
DIM   PRINT
END   READ
FOR   REM
GOSUB   RETURN
GOTO   RUN
IF   STOP
INPUT   TO
LET   TAB
LIST   THEN
                  USR

Words reserved in 8K, Extended and Disk versions. All the above
      plus:

```
        AND          ON
        CONT         OR
        DEF          OUT
        FN           POKE
        NOT          SPC
        NULL         WAIT
```

Words reserved in Extended and Disk versions. All the above plus:
```
        AUTO         LINE
        CONSOLE      kL              LLIST
        DEFDBL       LPRINT
        DEFINT       MOD
        DEFSNG       RENUM
        DEFSTR       RESUME
        DELETE       SPACE$
        EDIT         STRING$
        ELSE         SWAP
                     TROFF
        ERASE        TRON
        ERL          VARPTR
        ERR          WIDTH
        IMP          XOR
        INSTR
```

Words reserved in Disk. All the above plus:
```
        CLOSE        LSET
        DSKI$        MERGE
        DSKO$        MOUNT
        FIELD        NAME
        FILES        OPEN
        GET          PUT
        KILL         RSET
        LOAD            UNLOAD
```

APPENDIX A
ASCII CHARACTER CODES

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	−	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	^
009	HT	052	4	095	←
010	LF *Hex(0A)*	053	5	096	'
011	VT	054	6	097	a
012	FF *Hex(0C)*	055	7	098	b
013	CR *Hex 0D*	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	¦
039	'	082	R	125	
040	(	083	S	126	
041	)	084	T	127	DEL
042	*	085	U		

LF=Line Feed   FF=Form Feed   CR=Carriage Return   DEL=Rubout

Using ASCII codes -- the CHR$ function.

CHR$(X) returns a string whose one character is that
with ASCII code X. ASC(X$) converts the first character of
a string to its ASCII decimal value.

One of the most common uses of CHR$ is to send a
special character to the user's terminal. The most often
used of these characters is the BEL (ASCII 7). Printing
this character will cause a bell to ring on some terminals
and a beep on many CRT's. This may be used as a preface to
an error message, as a novelty, or just to wake up the user
if he has fallen asleep. Example:

PRINT CHR$(7);

Another major use of special characters is on those
CRT's that have cursor positioning and other special
functions (such as turning on a hard copy printer). For
example, on most CRT's a form feed (CHR$(12)) will cause the
screen to erase and the cursor to "home" or move to the
upper left corner.

Some CRT's give the user the capability of drawing
graphs and curves in a special point-plotter mode. This
feature may easily be taken advantage of through use of
Altair BASIC's CHR$ function.

APPENDIX B
LOADING AND INITIALIZING BASIC

A. Loading BASIC from paper tape or cassette.

     This appendix details the procedure for loading BASIC
in 4K, 8K and Extended versions from paper tape or tape
cassette. For instructions on loading Disk BASIC, see
appendix *H*.

     The programs below are entered into memory through the
front panel switches.   Rather than specify the switch
positions as "up" and "down", it is convenient to denote the
up position as 1 and the down position as 0. Taken in
groups of three, then, the switches can represent octal
digits.   To save space, the switch positions in the
following loader program listings are shown in   octal
notation.   The leftmost two switches in an 8 bit set are
represented by the first digit, the next three by the second
digit and the low-order three switches by the last digit.

     For example, if we wish to enter octal 315 on the data
switch register, the switches would have the following
positions:

     7   6   5   4   3   2   1   0
    up   up   down   down   up   up   down   up
        3   1   5

For data entry, only the rightmost 8 switches of the 16
switches on the ALTAIR 8800 front panel switch register are
used. All 16 switches would be used to enter a memory
address.

     The following is the procedure for loading BASIC from
paper tape or cassette.

1.   Turn the power switch on.

2.   Raise the STOP switch and RESET switch simultaneously

3.   Switch the terminal to LINE

4.   Enter one of the following programs on the front panel
     switches.   The 88-MBL Multi-Boot Loader PROM contains
     the necessary loader programs, so it is not necessary to
     enter a loader from the front panel if it is installed.
     Refer to the 88-MBL manual for more information.

    a.   loading from paper tape with the SIO board (REV 1)

Octal Address	Octal Data
000	041
001	302
002	0xx (17 for 4K, 37 for 8K, 77 for
003	061 Extended & Disk)
004	022
005	000
006	333
007	000
010	017
011	330
012	333
013	001
014	275
015	310
016	055
017	167
020	300
021	351
022	003
023	000

    b.   loading from cassette

Octal Address	Octal Data
000	041
001	302
002	0xx (17 for 4K, 37 for 8K, 77 for
003	061 Extended and Disk)
004	022
005	000
006	333
007	006
010	017
011	330
012	333
013	007
014	275
015	310
016	055
017	167
020	300
021	351
022	003
023	000

    c. loading with the 88 PIO board

Octal Address	Octal Code
000	041
001	302
002	0xx (17 for 4K, 37 for 8K, 77 for
003	061 Extended and Disk)
004	023
005	000
006	333
007	004
010	346
011	001
012	310
013	333
014	005
015	275
016	310
017	055
020	167
021	300
022	351
023	003
024	000

    d. loading with the 2SIO board

Octal Address	Octal Data
000	076
001	003
002	323
003	020
004	076
005	021 (=2 stop bits, 025=1 stop bit)
006	323
007	020
010	041
011	302
012	0xx (17for 4K, 37 for 8K, 77 for
013	061 Extended and Disk)
014	032
015	000
016	333
017	020
020	017
021	320
022	333
023	021
024	275
025	310
026	055
027	167

```
030          .        300
031                   351
032                   013
033                   000
```

e. loading with the 4PIO board

```
Octal Address        Octal Data
000                  257
001                  323
002                  040
003                  323
004                  041
005                  076
006                  054
007                  323
010                  040
011                  041
012                  302
013                  0xx  (17 for 4K, 37 for 8K, 77 for
014                  061   Extended and Disk)
015                  033
016                  000
017                  333
020                  040
021                  007
022                  330
023                  333
024                  041
025                  275
026                  310
027                  055
030                  167
031                  300
032                  351
033                  014
034                  000
```

f. Loading with the High Speed Tape Reader

```
Octal Address        Octal Data
000                  257
001                  323
002                  044
003                  323
004                  045
005                  323
006                  046
007                  057
010                  323
```

```
        011              047
        012              076
        013              014
        014              323
        015              044
        016              076
        017              004
        020              323
        021              046
        022              323
        023              047
        024              041
        025              302
        026              0xx  (17 for 4K, 37 for 8K, 77 for
        027              061   Extended and Disk)
        030              047
        031              000
        032              333
        033              044
        034              346
        035              100
        036              310
        037              333
        040              045
        041              275
        042              310
        043              055
        044              167
        045              300
        046              351
        047              027
        050              000
```

To enter these programs,

1.   Put switches 0 to 15 in the down positions

2.   Raise EXAMINE

3.   Put the data for address zero in switches 0 through 7.

4.   Raise DEPOSIT

5.   Put the data for the next address in the switches

6.   Depress DEPOSIT NEXT

7.   Repeat steps 5 and 6 until the whole loader is toggled
     in

8.   Put switches 0 through 15 in the down position

9.   Raise EXAMINE

10. Check to see that the lights D0 through D7 show the
     data that should be in location 000. Light on =1, light
     off = 0. If the correct value is there, go to step 13,
     if not go to 11.

11. Put the correct value in the switches

12. Raise DEPOSIT

13. Depress EXAMINE NEXT

14. Repeat steps 10 through 13 to check the entire loader

15. If there were any mistakes, check the entire loader
     again to make sure they were corrected.

16. If a paper tape is being loaded, put it into the reader
     and make sure that it is positioned at the beginning of
     the leader. The leader is the section of tape at the
     beginning with 302 octal punched in each column. If an
     audio cassette is being loaded, put it in the cassette
     recorder and make sure it is fully rewound.

17. Lower switches 0 through 15

18. Raise EXAMINE

19. Enter the sense switch settings. See the table in
     section B.

20. If loading is through a SIOA, B or C or an 88PIO, turn
     on the tape reader and then depress RUN. If a cassette
     is being loaded, turn on the recorder, put it in PLAY
     mode and wait 15 seconds. Then press RUN on the
     computer. If loading is through a 4PIO, 2SIO or High
     Speed Tape Reader, depress RUN and then start the read
     device.

21. Wait for the tape to read. Paper tape takes about 25
     minutes for Extended, 12 minutes for 8K and 6 minutes
     for 4K. Cassettes take about 8 minutes for Extended, 4
     minutes for 8K and 2 minutes for 4K. Do not move any of
     the switches while the tape is being read.

22. If a loading error occurs, the loading procedure must
     start over from step 1. See section C below for error
     conditions.

23. When the tape is read, BASIC should start up and print
     MEMORY SIZE? See section D below for what to do next.

24. If BASIC will not load from cassette, the ACR module
     may need realignment. The Input Test Program described
     in the ACR Manual, pages 22 and 28 may be used to test
     the ACR.

B.   Sense Switch Settings

     Sense switches (switches A8 through A15) must be set
before tape or cassette loading begins. The settings depend
on the terminal and input interface boards in use. The low
order (rightmost) four switches contain the load board
setting and the high order four switches contain the
terminal board setting. In the table below, the setting is
given for each I/O board option. As above, the setting is
an octal number which signifies the switch positions. The
Terminal Switch and Load Switch columns show the switches
that are raised for each of the load and terminal device
options.

Device	Sense Switch Setting	Terminal Switches	Load Switches	Channels
2SIO (2 stop bits)	0	none	none	20, 21
2SIO (1 stop bit)	1	A12	A8	20, 21
SIO	2	A13	A9	0, 1
ACR	3	A13,A12	A9,A8	6, 7
4PIO	4	A14	A10	40, 41, 42, 43
PIO	5	A14,A12	A10,A8	4, 5
HSR	6	A14,A13	A10,A9	46, 47
non-standard terminal	14			
no terminal	15			

Examples:

     Input from audio cassette through ACR and CRT terminal
     through 2SIO with 1 stop bit.
     Switch   15 14 13 12 11 10 9   8
     Position   0   0   0   1   0   0   1   1

     Input from high speed paper tape reader, terminal
     through SIO.
     Switch   15 14 13 12 11 10 9   8
     Position   0   0   1   0   0   1   1   0

C. Error Detection

     The checksum loader turns on the Interrupt Enable light
on the front panel when a loading error occurs. The ASCII
code of the error letter is stored in location 0.   In
addition, the error letter is sent out over all the terminal
channels and so will appear on whatever terminal   is
connected to the terminal.   The error letters are as
follows:

     C checksum error. Bad tape data.
     M memory error. Data won't store properly.
          The address of the bad memory location is stored
          in locations 1 and 2.
     O overlay error. Attempt was made to load data on top
          of the loader.
     I invalid load device. Invalid setting on the
          sense switches.

D. Initialization Dialog

     Upon starting, BASIC prints

     MEMORY SIZE?

To this, the user responds by typing the number of bytes of
memory to be used by BASIC and BASIC programs. Remember
that the BASIC interpreter itself takes 3.4K in the 4K
version, 6.2K in 8K and 14.6K in Extended. If the response
is just a carriage return, BASIC will use all the memory it
can find, starting at location zero up to the last byte of
read/write memory. Then BASIC asks,

     TERMINAL WIDTH?

to which the user responds with the width of the printing
line of whatever output device is in use. Typing a carriage
return sets the terminal width to 72.   Extended and Disk
Altair BASIC set the terminal width through the WIDTH
command, so the TERMINAL WIDTH question is not asked at
initialization and an initial width of 72 is assumed. In
4K, the response to MEMORY SIZE? and TERMINAL WIDTH?   must
be less than 6 digits.

     At this point BASIC asks several questions about
mathematical functions. The functions may be kept if needed
or deleted to save space.   4K asks,

          SIN?   Answer Y to save SIN, SQR and RND
                  Answer N to delete SIN and see the

```
            next question
     SQR?   Y keeps SQR and RND
            N deletes SQR, asks next question
     RND?   Y keeps RND
            N deletes RND
```

8K and Extended BASIC ask,

```
     WANT SIN-COS-TAN-ATN?        Y keeps all four
                                  N deletes all four
                                  A deletes only ATN   AND DISK
                                  C (in extended) retains
                                    CONSOLE function.  Any
                                    other answer deletes
                                    CONSOLE.
```

Now BASIC prints,

```
     XXXX BYTES FREE

     ALTAIR BASIC VERSION 4.0
     [FOUR-K VERSION]
           or
     [EIGHT-K VERSION]
           or
     [EXTENDED VERSION]
     OK
```

BASIC is now in command level and is ready for use.

E.   Echo Routines.

     The Altair input/output channels work in a full-duplex
mode. This means that characters entered on an input/output
terminal will not, as a rule, be printed as they are entered
unless the computer is programmed to return them. The
following echo programs may be used to test the input/output
devices.   To test an input-only device, dump the echoed
characters on an output device or store them in memory for
later examination.   To test an output-only device, send the
echo characters through the front panel switches or send a
constant character.   Be sure to check the ready-to-receive
bit of the output terminal before attempting output. If the
echo program works, but BASIC does not, make sure the load
device's I/O board is strapped for 8 data bits and that the
ready-to-recieve bit is set properly on the terminal device.

```
88-PIO
OCTAL ADDRESS          OCTAL CODE
     001                  004
     002                  346
     003                  001
```

```
                    004                 312
                    005                 000
                    006                 000
                    007                 333
                    010                 005
                    011                 323
                    012                 005
                    013                 303
                    014                 000
                    015                 000

            2SIO
            OCTAL ADDRESS        OCTAL CODE
                    000                 076
                    001                 003
                    002                 323
                    003                 020  (flag ch.)
                    004                 076
                    005                 021  (=2 stop bits,
                    006                 323    025=1 stop bit)
                    007                 020
                    010                 333
                    011                 020
                    012                 017
                    013                 322
                    014                 010
                    015                 000
                    016                 333
                    017                 021  (data channel)
                    020                 323
                    021                 021
                    022                 303
                    023                 010
                    024                 000

            4PIO
            OCTAL ADDRESS        OCTAL CODE
                    000                 257
                    001                 323
                    002                 040
                    003                 323
                    004                 041
                    005                 323
                    006                 042
                    007                 057
                    010                 323
                    011                 043
                    012                 076
                    013                 054
                    014                 323
                    015                 040
                    016                 323
```

017	042
020	333
021	040
022	346
023	200
024	312
025	020
026	000
027	333
030	042
031	346
032	200
033	312
034	027
035	000
036	333
037	041
040	323
041	043
042	303
043	020
044	000

APPENDIX C
SPACE AND SPEED HINTS

A. Space Allocation

     The memory space required for a program depends, of
course, on the number and kind of elements in the program.
The following table contains information on the space
required for the various program elements.

Element   Space Required

Variables
  numeric   integer   5 bytes
             single precision   7 bytes in Extended and Disk
                                6 bytes in 4K and 8K
             double precision   11 bytes
             string   6 bytes

Arrays
  integer   (# of elements) * $\lceil 2 \rceil$ + $\lceil 6 \rceil$ + (# of dimensions) * 2   bytes
  single precision   4
  double precision   8
  string   3
  8K and 4K
   strings and floating pt. $\lfloor 6 \rfloor$ + $\lfloor 5 \rfloor$

Functions
  intrinsic   1 byte for the call   (2 bytes in Extended and Disk)
  user-defined 6 bytes for the definition

Reserved Words 1 byte each
                2 bytes for ELSE in Extended and Disk

Other Characters
                1 byte each

Stack Space
  active FOR
     loop   17 bytes in Extended and Disk,
                16 bytes in 4K and 8K
  active GOSUB 5 bytes
  parentheses   6 bytes each set
  temporary
     result   12 bytes in Extended and Disk
                10 bytes in 4K and 8K

BASIC itself takes about 3.4K in the 4K version, 6.2K in 8K, 14.6K in Extended and 20 K in Disk.

B. Space Hints

The space required to run a program may be significantly reduced without affecting exectuion by following a few of the following hints.

1. Use multiple statements per line. Each line has a 5 byte overhead for the line number, etc., so the fewer lines there are, the less storage is required.

2. Delete unnecessary spaces. Instead of writing

    10 PRINT X, Y, Z

use

    10 PRINTX,Y,Z

3. Delete REM statements to save 1 byte for REM and 1 byte for each character of the remark.

4. Use variables instead of costants, expecially when the same value is used several times. For example, using the constant 3.14159 ten times in a program uses 40 bytes more space than assigning

    10 P=3.14159

once and using P ten times.

5. Using END as the last statement of a program is not necessary and takes one extra byte.

6. Reuse unneeded variables instead of defining new variables.

7. Use subroutines instead of writing the same code several times.

8. Use the smallest version of BASIC that will run the program.

9. Use the zero elements of arrays. Remember the array dimensioned by

    100 DIM A(10)

has eleven elements, A(0) through A(10).

10.   In Extended and Disk, use integer variables wherever possible.

C. Speed Hints

1.   Deleting spaces and REM statements gives a small but significant decrease in execution time.

2.   Variables are set up in a table in the order of their first appearance in the program. Later in the program, BASIC searches the table for the variable at each reference. Variables at the head of the table take less time to search for than those at the end.   So, reuse variable names and keep the list of variables as short as possible.

3.   In 8K, Extended and Disk use NEXT without the index variable.

4.   8K, Extended and Disk have faster floating point arithmetic than 4K.   If space is not a limitation, use the larger versions.

5.   The math functions in 8K, Extended and Disk are faster than those in 4K.

6.   In the 4K and 8K versions, use variables instead of constants, especially in FOR loops and other code that must be executed repeatedly.

7.   In Extended and Disk use integer variables wherever possible.

APPENDIX D
MATHEMATICAL FUNCTIONS

1. Derived Functions

The following functions, while not intrinsic to ALTAIR
BASIC, can be calculated using the existing BASIC functions.

Function: BASIC equivalent:

SECANT   SEC(X) = 1/COS(X)
COSECANT   CSC(X) = 1/SIN(X)
COTANGENT   COT(X) = 1/TAN(X)
INVERSE SINE   ARCSIN(X) = ATN(X/SQR(-X*X+1))
INVERSE COSINE   ARCCOS(X) = -ATN X(X/SQR(-X*X+1))
                                   +1.5708
INVERSE SECANT   ARCSEC(X) = ATN(XSQR(X*X-1))
                                   +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT   ARCCSC(X) = ATN(1/SQR(X*X-1))
                                   +(SGN(X)-1)*1.5708
INVERSE COTANGENT   ARCCOT(X) = ATN(X)+1.5708
HYPERBOLIC SINE   SINH(X) = (EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE   COSH(X) = (EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT   TANH(X) = EXP(-X)/EXP(X)+EXP(-X))
                                   *2+1
HYPERBOLIC SECANT   SECH(X) = 2/(EXP(X)+EXP(=X))
HYPERBOLIC COSECANT   CSCH(X) = 2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT   COTH(X) = EXP(-X)/(EXP(X)-EXP(-X))
                                   *2+1
INVERSE HYPERBOLIC
SINE   ARCSINH(X) = LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC
COSINE   ARCCOSH(X) = LOG(X+SQR(X*X+-1))
INVERSE HYPERBOLIC
TANGENT   ARCTANH(X) = LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC
SECANT   ARCSECH(X) = LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC
COSECANT   ARCCSCH(X) = LOG((SGN(X)*
                                   SQR(X*X+1)+1)/X
INVERSE HYPERBOLIC
COTANGENT   ARCCOTH(X) = LOG((X+1)/(X-1))/2

^&2. Simulated Math Functions.\&

The following subroutines are intended for 4K BASIC users

who want to use the transcendental functions not built into
4K BASIC. The corresponding routines for these functions in
the 8K version are much faster and more accurate. The REM
statements in these subroutines are given for documentation
purposes only, and should not be typed in because they take
up a large amount of memory. The following are the
subroutine calls and their 8K equivalents:

8K EQUIVALENT	4K SUBROUTINE CALL
P9=X9^Y9	GOSUB 60030
L9=LOG(X9)	GOSUB 60090
E9=EXP(X9)	GOSUB 60160
C9=COS(X9)	GOSUB 60240
T9=TAN(X9)	GOSUB 60280
A9=ATN(X9)	GOSUB 60310

The unneeded subroutines should not be typed in.   Please
note which variables are used by each subroutine. Also note
that TAN and COS require that the SIN function be retained
when BASIC is loaded and initialized.

```
60000 REM EXPONENTIATION: P9=X9^Y9
60010 REM NEED: EXP, LOG
60020 REM VARIABLES USED: A9,B9,C9,E9,L9,P9,X9,Y9
60030 REM P9 =1 : E9=0 : IF Y9=0 THEN RETURN
60040 IF X9<0 THEN IF INT(Y9)=Y9 THEN P9=1-2*Y9+4*INT(Y9/2)
      : X9=-X9
60050 IF X9<>0 THEN GOSUB 60090 : X9=Y9*L9 : GOSUB 60160
60060 P9=P9*E9 : RETURN
60070 REM NATURAL LOGARITHM: L9=LOG(X9)
60080 REM VARIABLES USED: A9,B9,C9,E9,L9,X9
60090 E9=0 : IF X9<=0 THEN PRINT "LOG FC ERROR"; : STOP
60100 A9=1: B9=2: C9=.5: REM THIS WILL SPEED THE FOLLOWING
60110 IF X9>=A9 THEN X9=C9*X9 : E9=E9+A9 : GOTO 60100
60120 X9=(X9-.707107)/(X9+.7077107) : L9=X9*X9
60130 L9=(((.598979*L9+.961471)*L9+2.88539)*X9+E9-.5)*
      .693147
60135 RETURN
60140 REM EXPONENTIAL : E9=EXP(X9)
60150 REM VARIABLES USED: A9,E9,L9,X9
60160 L9=INT(1.4427*X9)+1 : IF L9<127 THEN 60180
60170 IF X9>0 THEN PRINT "EXP OV ERROR"; : STOP
60175 E9=0 : RETURN
60180 E9=.693147*L9-X9 : A9=1.32988E-3-1.41316E-4*E9
60190 A9=((A9*E9-8.30136E-3)*E9+4.16574E-2)*E9
60195 E9=((A9-.166665)*E9-1)*E9+1 : A9=2
60197 IF L9<=0 THEN A9=.5 : L9=-L9 : IF L9=0 THEN RETURN
60200 FOR X9=1 TO L9 : E9=A9*E9 : NEXT X9 : RETURN
60210 REM COSINE: C9=COS(X9)
60220 REM N.B. SIN MUST BE RETAINED AT LOAD-TIME
60230 REM VARIABLES USED: C9,X9
```

```
60240 C9=SIN(X9+1.5708) : RETURN
60250 REM TANGENT: T9=TAN(X9)
60260 REM NEEDS COS. (SIN MUST BE RETAINED AT LOAD-TIME)
60270 REM VARIABLES USED: C9,T9,X9
60280 GOSUB 60240 : T9=SIN(X9)/C9   : RETURN
60290 REM ARCTANGENT : A9=ATN(X9)
60300 REM VARIABLES USED: A9,B9,C9,T9,X9
60310 T9=SGN(X9): X9=ABS(X9):C9=0: IF X>1 THEN C9-1: X9=1/X9
60320 A9=X9*X9  : B9=((2.86623E-3*A9-1.61657E-2)*A9
         +4.29096E-2)*A9
60330 B9=((((B9-7.5289E-2)*A9+.106563)*A9-.1142089)*A9+.199936)*A9
60340 A9=((B9-.333332)*A9+1)*X9 : IF C9=1 THEN A9=1.5708-A9
```

APPENDIX E
BASIC AND ASSEMBLY LANGUAGE

All versions of Altair BASIC have provisions for
interfacing with assembly language routines. The USR
function allows Altair BASIC programs to call assembly
language subroutines in the same manner as BASIC functions.

The first step in setting up a machine language
subroutine for an Altair BASIC program is to set aside
memory space. When BASIC asks, MEMORY SIZE? during
initialization, the response should be the size of memory
available, minus the amount needed for the assembly language
routine. BASIC uses all the bytes it can find from location
zero up, so only the topmost locations in memory can be used
for user supplied routines. If the answer to the MEMORY
SIZE? question is too small, BASIC will ask the question
again until it gets all the memory it needs. See Appendix
C.

The assembly language routine may be loaded into memory
from the front panel switches or from a BASIC program by
means of the POKE statement.

The starting address of the assembly language routine
goes in USRLOC, a two byte location in memory which varies
from version to version. USRLOC for 4K and 8K Altair BASIC
version 4.0 is 111 OcTAL . In Extended and Disk, USRLOC
need not be known explicitly since it is defined
automatically by DEFUSR. See section 5-3b. The function
USR calls the routine whose address is in USRLOC.
Initially, USRLOC contains the address of ILLFUN, the
routine which gives the FC or ILLEGAL FUNCTION CALL error,
which is what happens if USR is called with no assembly
language routine having been loaded.

When USR is called, the stack pointer is set up for 8
levels (16 bytes) of stack storage. If more stack space is
needed, BASICs stack can be saved and a new stack set up for
use by the assembly language routine. BASIC's stack must be
restored, however, before returning from the user routine.

All memory and all the registers can be changed by a
user's assembly language routine. Of course, memory
locations within BASIC ought not to be changed, nor should
more bytes be popped off the stack than were put on it.

USR is called with a single argument. The assembly
language routine can retrieve this argument by calling the
routine whose address is in locations 4 and 5 decimal. The

low-order byte of the address is in 4 and the high-order in
5. In 4K and 8K, this routine (DEINT) stores the argument
in the register pair [D,E]. In Extended, the argument is
passed in pair [H,L]. The argument is truncated to integer
in 4K and 8K, and if it is not in the range -32768 to 32767,
an FC error occurs. In extended, the register pair [H,L]
contains a pointer to the Floating Point Accumulator where
the argument is stored (see section 5-3b. for more
information).

To pass a result back from an assembly language
routine, load the value in register pair [A,B] in 4K and 8K,
or [H,L] in Extended. This value must be a signed, 16 bit
integer as defined above. Then call the routine whose
address is in locations 6 and 7. If this routine is not
called, USR(X) returns X. To return to BASIC, then, the
assembly language routine executes a RET instruction.

Assembly language routines can be written to handle
interrupts. Locations 56, 57 and 58 are used to hold a JMP
instruction to a user supplied interrupt handling routine.
Location 56 initially holds a RET, so it must be set up by
the user or an interrupt will have no effect.

All interrupt handling routines should save the stack,
registers A-L and the PSW. They should also reenable
interrupts before returning since an interrupt automatically
disables all further interrupts once it is received.

There is only one way to call an assembly language
routine in 4K and 8K, but this does not limit the programmer
to only one assembly language routine. The argument of USR
can be used to designate which routine is being called. In
8K, additional arguments can be passed through the use of
POKE and values may be passed back by PEEK.

In Extended and Disk BASIC, up to ten routines may be
called with the USR0 - USR9 functions. For more information
on this feature, see section 5-3b.

APPENDIX F
USING THE ACR INTERFACE

NOTE

The cassette features , CLOAD and CSAVE , are only
present in 8K Altair BASICs which are distributed on
cassette, and in Extended and Disk versions. 8K
BASIC on paper tape will give the user about 250
additional bytes of free memory, but it will not
recognize the CLOAD or CSAVE commands.

Programs may be saved on cassette tape by means of the
CSAVE command. CSAVE may be used in either direct or
indirect mode, and its format is as follows:

CSAVE <string expression>

The program currently in memory is saved on cassette under
the name specified by the first character of the STRING
<expression>. CSAVE writes through channel 7 when the Write
Buffer Empty bit (bit 7) of channel 6 is low. After CSAVE
is completed, BASIC always returns to command level.
Programs are written on tape in BASIC's internal
representation. Variable values are not saved on tape,
although an indirect mode CSAVE does not affect the variable
values of the program currently in memory. The number of
nulls (see NULL command) has no affect on the operation of
CSAVE. Before using CSAVE, turn on the cassette recorder,
make sure the tape is in the proper position and put the
recorder in RECORD mode.

NOTE: THE PROGRAM NAMED A IS SAVED BY :SAVE "A".

Programs may be loaded from cassette tape by means of
the CLOAD command, which has the same format as CSAVE. The
effect of CLOAD is to execute a NEW command, clearing memory
and all variable values, and loading the specified file into
memory. When done reading and loading, BASIC returns to
command level. CLOAD reads a byte from channel 7 when the
Read Data Ready bit (bit 0) in channel 6 is low. Reading
continues until 3 consecutive zeros are read. BASIC will
not return to command level after a CLOAD if it could not
find the requested file or if the file was found but did not
end with 3 zeros. In that case, the computer will continue
to search until it is stopped and restarted at location 0.

In the 8K cassette and Extended versions of ALTAIR
BASIC, data may be read and written with the CSAVE* and
CLOAD* commands. The formats are as follows:

CSAVE*<array variable name>

and

CLOAD*<array variable name>

See section 2-4d for a discussion of CSAVE* and CLOAD* for
array data.

CLOAD?<string expression> compares the program
currently in memory with the specified file on cassette. If
the two files match, BASIC prints OK. If not, BASIC prints
NO GOOD.

Data may also be read from and written on cassette in
the paper tape version of 8K Altair BASIC. To write data,
execute a WAIT 6,128 statement to check for the Write Buffer
Empty bit and then write with an OUT 7,<byte> statement. To
read, execute a WAIT 6,1 to check for Read Data Ready and
then read with an INP(7). The end of a block of data may be
conveniently designated by a special character. Data should
be stored in array form since there is no time during
reading and writing for computation.

APPENDIX G
CONVERTING BASIC PROGRAMS
NOT WRITTEN FOR THE ALTAIR COMPUTER

Though implementations of BASIC on different computers are in many ways similar, there are some incompatibilities between ALTAIR BASIC and the BASIC used on other computers.

1) Strings.

A number of BASICs require the length of strings to be declared before they are used. All dimension statements of this type should be removed from the program. In some of these BASICs, a declaration of the form DIM A$(I,J) declares a string array of J elements each of which has a length I. Convert DIM statements of this type to equivalent ones in Altair BASIC: DIM A$(J). Altair BASIC uses " + " for string concatenation, not " , " or " & ." ALTAIR BASIC uses LEFT$, RIGHT$ and MID$ to take substrings of strings. Some other BASICs use A$(I) to access the Ith character of the string A$, and A$(I,J) to take a substring of A$ from character position I to character position J. Convert as follows:

```
OLD                      NEW
A$(I)                    MID$(A$,I,1)
A$(I,J)                  MID$(A$,I,J-I+1)
```

This assumes that the reference to a subscript of A$ is in an expression or is on the right side of an assignment. If the reference to A$ is on the left hand side of an assignment, and X$ is the string expression used to replace characters in A$, convert as follows :

```
In 4K and 8K
OLD                      NEW
A$(I)=X$                 A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1)
A$(I,J)=X$               A$=LEFT$(A$,I-1)+X$+MID$(A$,J+1)

Extended and Disk
OLD                      NEW
A$(I)=X$                 MID$(A$,1,1)=X$
A$(I,J)=X$               MID$(A$,I,J-I+1)=X$
```

2) Multiple assignments.

Some BASICs allow statements of the form:

        500 LET B=C=0

This statement would set the variables B and C to zero.   In
8K Altair BASIC this has an entirely different effect.   All
the " = " signs to the right of the first one would be
interpreted as logical comparison operators.   This would set
the variable B to -1 if C equaled 0.   If C did not equal   0,
B would be set to 0.   The easiest way to convert statements
like this one is to rewrite them as follows.

        500 C=0:B=C

3) Some BASICs use " \ " instead of " : " to delimit
multiple statements on a line. Change each " \ " to " : "
in the program.

4) Paper tapes punched by other BASICs may have no nulls at
the end of each line, instead of the three per line
recommended for use with Altair BASIC. To get around this,
try to use the tape feed control on the Teletype to stop the
tape from reading as soon as Altair BASIC prints a carriage
return at the end of the line. Wait a moment, and then
continue feeding in the tape. When reading has finished, be
sure to punch a new tape in Altair BASIC's format.

    A program for converting tapes to Altair BASIC's format
was published in MITS Computer Notes, November 1976, p.   25.

5) Programs which use the MAT functions available in some
BASICs will have to be re-written using FOR...NEXT loops to
perform the appropriate operations.

APPENDIX H
DISK INFORMATION

Format of Altair Floppy Disk

Track Allocation:

```
Tracks        Use
0-5  . Disk BASIC memory image.
6-69  Space for either random or sequential files.
 70   Directory track. See below.
71-76 Space for sequential files only.
```

Format of DISK BASIC Memory Image (Tracks 0-5):

BASIC is loaded starting at track 0 sector 0 then track 0
sector 1, etc.   Each sector contains 128 bytes of BASIC.
The first 128 bytes are loaded first, second 128 second,
etc.

Sector format (Tracks 0-5):

```
Byte  Use
0     Track Number+128 decimal.
1-2   Sixteen bit address of the next
      higher byte of memory than the highest memory location
        saved on this sector.
3-130 128 bytes of BASIC.
131   255 decimal stop byte.
132   Checksum - sum of bytes 3-130 with no carry in 8 bits.
```

Sector format (Tracks 6-76):

```
Byte  Use
0     Most Significant Bit always on.
        Contains track number plus 200 octal.
1     Sector number * 17 MOD 32.
2     File number in directory. Zero file number means
      that the sector is not part of any file. If the
      sector is the first file of a group of 8 sectors
      0 means the whole group of 8 sectors is free.
```

3   Number of data bytes written (0 to 128) . Always
        128 for random files. (Except for the random file
           index blocks in which case this byte indicates how many
        groups are allocated to the file.)
4   Checksum. The sum of all the data on the sector
        except for the track number, the sector
        number and the terminating 255 byte.
5,6   Pointer to the next group of data. This is set up for
        random files and sequential files, and is even valid
        in the middle of a group. If it is zero it means there
        is no more data in the file. The track is the first byte
        and the sector number is the second byte.
7-134 Data
135   A 255 (octal 377) to make sure the right number
        of data bytes were read.
136   Unused.

Directory Track (70) Format:

        Each sector of the directory (which is all of track 70)
is composed of up to 8 file name slots, 16 bytes per slot.
Each slot can contain a file name (8 bytes), a link to the
start of file data (2 bytes), and a byte which specifies the
mode of a file (Random=4, Sequential=2).   The remaining 5
bytes are not currently used. Any slot which has the first
filename byte equal to zero contains a file which has been
deleted.   If the first byte of a slot is a 255 , it is the
last slot currently in use in the directory.   Slots beyond
the "stopper" are garbage. File numbers are calculated by
multiplying the sector number of the directory track the
file is in by 8 and adding the position of the slot in the
sector (0-7) plus 1.

                              NOTE

    The ith logical sector on a track is actually mapped
    to the i*17 MOD 32 physical sector to improve
    latency in BASIC I/O operations.

                Format of Random Files

Each random file starts with two random index blocks.   The
"number of data bytes" field in the first block indicates
how many groups are currently allocated to this random file.
The next 256 bytes in the two random index blocks give the
location of each group in the random file in order of their
position in the file.   The upper two bits give the group
number , and the lower six bits give the track number - 6.

Assembly Code to Read and Write a Sector

The following code has been provided to help users write their own assembly language subroutines to read and write data on the floppy disk. It is assumed that the disk being used has already been enabled and positioned to the correct track. Two data bytes are always read or written at a time so that the CPU can keep up with the data rate (32 microseconds/byte) of the floppy disk. After two bytes are read or written, the CPU re-synchronizes with the next 'byte ready' status from the floppy disk controller.

```
; CALL WITH NUMBER OF DATA BYTES TO WRITE IN [A]
; AND POINTER TO DATA BUFFER IN [H,L]
; ALL REGS DESTROYED.

DSKO:   MOV     C,A             ;SAVE # OF BYTES IN C
        MVI     A,136           ;CALCULATE NUMBER OF ZEROS TO WRITE
        SUB     C               ;SUBTRACT THE NUMBER OF DATA BYTES
        MOV     B,A             ;NUMBER OF ZEROS+1
        CALL    SECGET          ;LATENCY
        MVI     A,128           ;ENABLE WRITE WITHOUT SPECIAL CURRENT
        OUT     9
;
; CALL WITH [B]=NUMBER OF ZEROS [C]=NUMBER OF DATA BYTES
; AND [H,L] POINTING AT OUTPUT DATA
;
OHLDSK:     MVI     D,1             ;SETUP A MASK (READY TO WRITE)
        MVI     A,128           ;HIGH BIT (D7) ALWAYS ON IN FIRST BYTE
        ORA     M               ;OR ON DATA BYTE
        MOV     E,A             ;SAVE FOR LATER
        INX     H               ;INCREMENT BUFFER POINTER
NOTYTD:     IN      8           ;GET WRITE DATA READY STATUS
        ANA     D               ;TEST STATUS BIT
        JNZ     NOTYTD          ;NOT READY TO WRITE, WAIT
        ADD     E               ;ADD BYTE WE WANT TO SEND TO ZERO
        OUT     10              ;SEND THE BYTE
        MOV     A,M             ;GET NEXT BYTE TO SEND
        INX     H               ;MOVE BUFFER POINTER AHEAD
        MOV     E,M             ;GET NEXT DATA BYTE
        INX     H               ;MOVE BUFFER POINTER AHEAD AGAIN
        DCR     C               ;DECREMENT COUNT OF CHARS TO SEND
        JZ      ZRLOP           ;IF DONE, QUIT & GO TO ZRLOP
        DCR     C               ;DECREMENT COUNT OF CHARS AGAIN
        OUT     10              ;SEND THIS BYTE
        JNZ     NOTYTD          ;STILL MORE CHARS, DO THEM.
ZRLOP:      IN      8           ;GET READY TO WRITE
        ANA     D               ;IS IT READY
        JNZ     ZRLOP           ;IF NOT, LOOP
        OUT     10              ;KEEP SENDING FINAL BYTE
        DCR     B               ;DECREMENT COUNT OF BYTES TO SEND
```

```
          JNZ     ZRLOP               ;KEEP WAITING
          EI                          ;RE-ENABLE INTERRUPTS
          MVI     A,8                 ;UNLOAD HEAD
          OUT     9                   ;SEND COMMAND
          RET                         ;DONE

; DISK INPUT ROUTINE. ENTER WITH POINTER
; OF 137 BYTE BUFFER IN [H,L]. ALL REGS DESTROYED.
DSKI: CALL    SECGET               ;POINT TO RIGHT SECTOR
      MVI     C,137                ;GET # OF CHARS TO READ
READOK:       IN      8                   ;GET DISK STATUS
      ORA     A                    ;READY TO READ BYTE
      JM      READOK
      IN      10                   ;READ THE STUFF
      MOV     M,A                  ;SAVE IN BUFFER
      INX     H                    ;BUMP DESTINATION POINTER
      DCR     C                    ;LESS CHARS
      JZ      RETDO                ;IF OUT OF CHARS, RETURN
      DCR     C                    ;DECREMENT COUNT OF CHARS
      NOP                          ;DELAY INTO NEXT BYTE
      IN      10                   ;GET NEXT BYTE
      MOV     M,A                  ;SAVE BYTE IN BUFFER
      INX     H                    ;MOVE BUFFER POINTER
      JNZ     READOK               ;IF CHARS STILL LEFT, LOOP BACK
RETDO:        EI                          ;RE-ENABLE INTERRUPTS
      MVI     A,8                  ;UNLOAD HEAD
      OUT     9                    ;SEND COMMAND
      RET

SECGET:       MVI     A,4                 ;LOAD THE HEAD
      OUT     9
      DI                           ;DISABLE INTERRUPTS
SECLP2:       IN      9                   ;GET SECTOR INFO
      RAR                          ;FIX UP SECTOR #
      JC      SECLP2               ;IF NOT, KEEP WAITING
      ANI     31                   ;GET SECTOR #
      CMP     E                    ;IS IT THE ONE WE WANTED
      JNZ     SECLP2               ;TRY TO FIND IT
      RET
```

The Disk PROM Bootstrap Loader

The Disk bootstrap loader PROM must be installed in the
highest position on the PROM board and the PROM board must
be strapped at the proper address. The proper position is
the PROM IC socket on the opposite side of the board from
the black finned heat sink. The black dot or '1' on the
PROM should be in the upper left corner. The address
jumpers on the PROM board must be in the '1' position.

To use the Disk bootstrap loader, turn the computer's power
on.   Raise RESET and STOP simultaneously. Lower RESET and
then STOP. EXAMINE location 177400 (address switches A15-A8
up, rest down) and then set the sense switches for the
terminal I/O board as explained in Appendix B. Depress the
RUN switch. BASIC should print (or display):

MEMORY SIZE?

For the rest of the initialization procedure, see below.

     Using the Cassette and Paper Tape Bootstraps

If the Disk Bootstrap PROM is not in use, a paper tape or
cassette program must be loaded which then reads in BASIC
from the disk. This is done by following the procedure
below:

1. Key in the applicable paper tape or cassette bootstrap
    loader   from   the   listings in Appendix B.   Make
    location 2=077 octal.   Set the sense switches for the
    terminal

2. Start the paper tape or cassette (labeled DISK LOADER)
    reading, and then   start the computer as in the
    instructions for loading BASIC   from paper tape from
    cassette as given in   Appendix B.

BASIC should respond:

MEMORY SIZE?

For the rest of the initialization procdure, see below.

               Disk Initialization Dialog

The initialization dialog has been expanded to allow the
user to select the proper amount of memory needed to use the
disk(s) on the system. After the the MEMORY SIZE question
is answered, BASIC will ask:

HIGHEST DISK NUMBER?

The user should answer with the highest physical disk
address in the system or with carriage return to default to
0. Each additional disk uses 40 bytes of memory.

Example:

HIGHEST DISK NUMBER? 1

BASIC next asks how many files are to be OPEN at one time in
the program. This number includes both random and
sequential files. If the user types carriage return, the
default is zero. Each file allocated requires 138 bytes for
buffer space. Example:

HOW MANY FILES? 2

Finally, BASIC asks how many random files are to be OPEN at
one time. The amount of memory allocated is the answer*257.
This memory space is used to keep track of the location on
the floppy disk where groups of a random file reside. Thus,
the total memory required for each random file is
138+257=395 bytes. Example:

HOW MANY RANDOM FILES? 1

A typical dialog might appear as follows:

MEMORY SIZE? <carriage return>
HIGHEST DISK NUMBER? <carriage return>
HOW MANY FILES? 2 <carriage return>
HOW MANY RANDOM FILES? 1 <carriage return>

xxxxx BYTES FREE
Altair BASIC REV. 4.0
[DISK EXTENDED VERSION]
COPYRIGHT 1976 BY MITS INC.

OK

APPENDIX I

THE PIP UTILITY PROGRAM

A BASIC Utility program has been provided to perform such
such common functions as printing directories, initializing
disks, copying disks etc.

NOTE

>   Some of the PIP commands (LIS, DIR) require that one
>   <file number> be configured during the Disk BASIC
>   initialization dialog. This is done by answering
>   the "HOW MANY FILES?" question with a value greater
>   than zero. If an attempt is made to perfrom a LIS
>   or   DIR   without   following   this procedure, a
>   BAD FILE NUMBER error will occur.

Once the BASIC disk has been mounted, type the following
command:

>   RUN "PIP"<carriage return>
>   (PIP will type)
>   *

PIP is now ready to accept commands. To exit PIP, type a
carriage return to the prompt asterisk. To initialize the
floppy disk in drive 0, type:

>   *INI0

PIP will type "DONE" when it is finished. Any disk number
may be substituted for the 0 in the above command and PIP
will format the disk in that drive. Any previous files on
the disk initialized will be lost. If you wish to use blank
disks with Disk BASIC, they must be initialized in this
fashion before they can be MOUNTed.

NOTE

>   DO NOT INITIALIZE THE DISK WITH DISK EXTENDED BASIC
>   ON IT. THIS WILL WIPE OUT ALL THE FILES PROVIDED ON
>   THE DISK.

Printing a Directory

Giving PIP the command:

     *DIR<disk number>

prints out a directory of the files on the specified disk.
The name of each file is printed, along with the file's
"mode" (S for sequential, R for random), and the starting
track and sector number of the first block in the file.

     SRT<disk number>

prints a sorted directory of the files on the specified
disk.

LISting Sequential Files

The LIS command is used to list the contents of a sequential
data file on the terminal:

Syntax:

     LIS<disk number>,<file name>

Example:

     *LIS0,PIPA   user types
     7 CLEAR 1000   computer prints
          .
          .
          .

     *

COPying Disks

The COP command is used to copy a disk placed in one drive
to a disk on another drive. Neither disk need be MOUNTed
for the COP command to work properly.

Syntax:

     COP<old disk number>,<new disk number>

Before the copy is done, PIP verifies the actionn by printing the following massage:

FROM<disk number>TO<disk number>

Typing Y followed by a carriage return causes execution to proceed.   Any other responce aborts the command.   Example: *COP0,1 FROM 0 TO 1? Y<CARRIAGE return> DONE *

The DAT command

The DAT command is used to dump out a particular sector of the disk in octal.

Syntax:

DAT<disk number>

When the DAT command is issued, PIP asks for the numbers of the track and sector to be dumped. Example: *DAT0
                    TRACK?   0 SECTOR?   0 000 000 000 000 000   000
000 000 000 000 000 000 000 etc.

The CNV command

CNV converts disks written under Altair BASIC version 3.4 and   3.3 to a format useable by version 4.0.   The format of the command is as follows:.

CNV<disk number>

CNV makes sure that the next to last byte of each sector is 255.

Other Programs Provided on the System Disk

Program Name   Use
STARTREK   Plays game based on TV series.

APPENDIX J
BASIC TEXTS

     Below are a few of the many texts that may be helpful
in learning BASIC.
1) BASIC PROGRAMMING, John G.   Kemeny,   Thomas   E.   Kurtz,
1967, 145pp.
2) BASIC, Albrecht, Finkel and Brown, 1973
3) A GUIDED TOUR OF COMPUTER PROGRAMMING IN BASIC, Thomas A.
Dwyer and Michael S. Kaufman; Boston: Houghton Mifflin
Co., 1973

Books numbered 1 and 2 may be obtained from:

           People's Computer Company
           P.O. Box 310
           Menlo Park, California   94025

They also have other books of interest, such as:

101 BASIC GAMES, David Ahl, Ed., 1974, 250pp.
WHAT TO DO AFTER YOU HIT RETURN or PCC's FIRST BOOK OF
COMPUTER GAMES
COMPUTER LIB AND DREAM MACHINES, Theodore H. Nelson, 1974,
186pp.

APPENDIX K

USING Altair BASIC ON THE
INTELLEC* 8/MOD 80 AND MDS SYSTEMS.

This appendix covers procedures for loading and operating Altair BASIC on Intellec and MDS development systems.

A. Loading BASIC. To load Altair BASIC, put the hex paper tape of BASIC in the system reader device. Now enter the System and assign the CONSOLE I/O device as desired (see Section 4.2.1 of the Intellec 8/Mod 80 Operator's Manual). Now read in BASIC with the following R command.

    .R(Cr)

The BASIC tape will be loaded into memory and the system monitor will type a period on the CONSOLE device. If you are only using contiguous RAM memory below the system monitor (3800H) or are using BASIC on a MDS System, proceed to step 2. If you have RAM memory above the PROM Intellec monitor which you wish BASIC to use for program and variable storage, you must patch the two locations known as INTLOC to point to the bottom (lowest address) of memory. The is most easily accomplished by using the System Monitor S command. INTLOC is given below under "Memory Requirements."

    .SXXXX 00 40 (Cr)

The above S command would make INTLOC point to RAM, starting at 16K.

NOTE

If you are using RAM above 16K for program and variable storage and have patched INTLOC, retain all the math functions at initialization time (see Appendix B). Essentially, this means that the WANT SIN-COS-TAN-ATN? questions asked by BASIC's initialization dialog should be answered by a Y(Cr). Also, you must answer the MEMORY SIZE? question with the highest decimal or RAM address in your system.

Start BASIC by giving the monitor GOTO command

.G0000<carriage return>

NOTE

Once BASIC has been started, it may always be
restarted by depressing the RESET switch on the
Intellec 8 console.

When BASIC types MEMORY SIZE?, Typing carriage return will
cause BASIC to use all the RAM memory it can find above the
end of BASIC. Otherwise, if you wish to specify an exact
amount of memory, type the decimal address of the highest
byte of memory in the computer and type carriage return.

B.   BASIC I/O.

The system devices used for terminal I/O in BASIC are
CI, CO and CSTS.

C.   Saving and Loading Programs.

To save a program on paper tape, re-enter the PROM
monitor and reassign the CO device to the paper tape punch
or other output device. Then restart BASIC by using the
G0000 command and type LIST(Cr). The characters of the LIST
command will not be echoed, but the BASIC program currently
saved in memory will be put on the output device.

To load a program enter the system monitor, re-assign
CI to the input device where the program resides, and then
start BASIC with a G0000. When the program has been
completely read in, reassign CI to the user console. Then
re-enter BASIC with a G0000, and start the I/O device. The
program will be echoed on CO as it is read in.

D.   Memory Requirements

BASIC   uses   locations   0000H-0003H   and
0010H-approximately 19DFH in the 8K version, and 0010H-2F0EH
in the Extended version. For Intellec 8K and MDS 8K BASICs,
INTLOC is 6520 decimal. For MDS Extended, INTLOC is 14257
decimal.

E.   Calling Assembly Language Routines

USRLOC for 8K BASIC is 0055H. ADR(DEINT) is stored in
locations 0043H.   ADR(GIVACF) is stored in location 0045H.
In the Extended version these locations   contain   the
addresses of FRCINT and MAKINT, respectively. Interrupt
driven subroutines using RST 7 are not allowed in the
Intellec/MDS version of Altair BASIC. See Appendix C. for
further   information   on   calling   assembly   language
subroutines.

* Intellec is a registered trademark of the Intel
Corporation.

APPENDIX L
PATCHING BASIC'S I/O ROUTINES

BASIC's I/O routines may be changed to accommodate non-standard terminal equipment. After BASIC is loaded and before it has been initialized, location 71 contains a pointer to a list of addresses. These addresses contain the I/O routines of BASIC:

```
        ORG     71Q
        DW      IOLST              ;TWO  BYTE ADDRESS OF ADDRESS LIST
        .
        .
        .
IOLST:  DW      TRYOUT             ;ADDRESS OF OUTPUT ROUTINE
        DW      TRYIN              ;CHARACTER INPUT ROUTINE
        DW      ISCNTC             ;POLL FOR CONTROL/C CHECK
        DW      NEWSTT             ;FAST POLL FOR CONTROL/C CHECK
                                   ;8K AND LARGER ONLY
        DW      IN2SIO             ;ADDRESS OF INITIALIZATION
                                   ;ROUTINE FOR 2SIO BOARDS
        DW      IN4PIO             ;ADDRESS OF INITIALIZATION ROUTINE FOR
                                   ;4PIO BOARDS
        DW      LPTCOD             ;ADDRESS OF LPT ROUTINE (IN EXTENDED
                                   ;AND DISK ONLY.)
        DW      LPTCD2             ;2ND LPT ROUTINE
        DW      LPTCD3             ;3RD LPT ROUTINE
        DW      IOCHNL             ;ADDRESS OF I/O RESET LOCATION
                                   ;(IN EXTENDED AND DISK ONLY)
        .
        .
        .
TRYOUT: IN      0                  ;GET DEVICE STATUS
        ANI     200                ;AND OFF BIT 7
        JNZ     TRYOUT             ;WAIT UNTIL TERMINAL CAN OUTPUT
        POP     PSW                ;GET CHARACTER TO OUTPUT OFF STACK
        OUT     1                  ;TRANSMIT IT
        PUSH    PSW                ;SAVE CHARACTER BACK ON STACK
        NOP                        ;CHANGED TO "IN 41" FOR 4PIO BOARDS
        NOP
        POP     PSW                ;GET CHARACTER BACK OFF STACK
        RET                        ;ALL DONE WITH CHARACTER OUTPUT ROUTINE
        .
        .
        .
TRYIN:  IN      0                  ;GET TERMINAL STATUS
        ANI     1                  ;CHARACTER READY?
        JNZ     TRYIN              ;NO, KEEP WAITING
```

```
          IN      1                  ;READ IN THE CHARACTER
          ANI     127                ;GET RID OF PARITY BIT
          CPI     CONTO              ;CONTROL/O?
          RNZ                        ;RETURN IF NOT
          .
          .
          .
ISCNTC: IN        0                   ;READ TERMINAL STATUS

          ANI     11                 ;HAS THE TERMINAL A CHARACTER

                                     ;TO SEND?
          RNZ                        ;NO, RETURN
          .
          .
          .
;FOLLOWING ROUTINE IS IN 8K AND LARGER VERSIONS ONLY
;AND IS EXECUTED FOR EACH STATEMENT
NEWSTT: IN        0                   ;READ TERMINAL STATUS
          ANI     1                  ;TEST BIT 0
          CZ      CNTCCN             ;YES, SEE IF CHARACTER CONTROL/C
          .
          .
          .
IN2SIO: CPI       2*4                 ;IS IT 2SIO
          RNC                        ;NO, OTHER GO DIRECTLY TO SETIO
          ADI     21                 ;GET PROPER INITIALIZATION BYTE
          PUSH    PSW                ;SAVE IT
          MVI     A,3                ;INITIALIZE THE 2SIO
          CALL    DOIO20
          POP     PSW                ;GET BACK SECOND INITIALIZATION BYTE
          JMP     DOIO20             ;PROGRAM TO DATA AND STOP BITS
          .
          .
          .
IN4PIO: MVI       A,54Q               ;RESET FOR DATA TRANSFER
          DCR     M                  ;CHANNEL=22
          CALL    DOIO20
          .
          .
          .
LPTCOD: LDA       PRTFLG              ;SEE IF WE WANT TO TALK TO LPT
          ORA     A                  ;TEST BITS
          JZ      TTYCHR             ;IF ZERO THEN NOT
          POP     PSW                ;GET BACK CHAR
          PUSH    PSW
          CPI     9                  ;TAB
          JNZ     NOTABL             ;NO
MORSPL: MVI       A,32                ;GET SPACE
          OUTCHR                     ;SEND IT
          LDA     LPTPOS             ;GET CURRENT PRINT POSIT
          ANI     7                  ;AT TAB STOP?
```

```
                JNZ     MORSPL          ;GO BACK IF MORE TO PRINT
                POP     PSW             ;POP OFF CHAR
                RET                     ;RETURN
        NOTABL:
                POP     PSW              ;GET CHARACTER WE WANT TO PRINT
                PUSH    PSW
                CPI     13              ;IS IT CARRIAGE RETURN?
                CZ      PRINTW          ;FORCE OUT A LINE
                CPI     13              ;GET CONDITION CODES BACK
                JC      PPSWRT          ;IF FUNNY CONTROL CHARACTER
                                         ;(LF), DO NOTHING
                LDA     LPTPOS          ;WHERE ARE WE?
                CPI     LPTLEN-1        ;ARE WE AT END OF LINE?
                JNZ     NOTELP          ;NO, JUST SEND CHAR
                MVI     A,1             ;SET LPTLST=1 AND LPTPOS=0
                CALL    FINLP2
                DCR     A               ;MAKE SURE LPTPOS ZERO.
        NOTELP: INR     A
                STA     LPTPOS
        LPTWAT: IN      2
                ORI     245
                INR     A
                JNZ     LPTWAT
                POP     PSW
                OUT     3               ;SEND OUT CHAR
                RET                     ;RETURN
        ;THIS ROUTINE IS CALLED TO FORCE OUT A PARTIAL BUFFER
        ;FOR THE LINE PRINTER. IT ALSO RESETS PRTFLG SO ALL
        ;FURTHUR I/O GOES TO THE USER'S TERMINAL
        FINLPT: XRA     A               ;RESET PRINT FLAG SO OUTPUT
                STA     PRTFLG          ;GOES TO THE TERMINAL
                LDA     LPTPOS          ;SEE IF ANY LEFTOVERS MUST BE
                ORA     A               ;FORCED OUT
                RZ                      ;BY LOOKING AT LPTPOS
        ;THE ROUTINE PRINTW IS CALLED TO FORCE OUT A LINE CURRENTLY
        ;IN THE LINE PRINTER BUFFER. THE CARRIAGE RETURN/LINE FEED
        ;OUTPUT SUBROUTINE CALLS PRINTW
        PRINTW: IN      2               ;MAKE SURE LAST PRINT
                ORI     245
                INR     A
                JNZ     PRINTW          ;BIT
        ; SEE IF BUFFER MUST BE EMPTIED
                LDA     LPTPOS
                ORA     A               ;CHARACTERS IN THE BUFFER?
                JNZ     PRINTR          ;IF SO DON'T CLEAR THE BUFFER
                LDA     LPTLST          ;PRINT BLANK LINE.
                                         ;CHECK IF PRINT WAS LAST
                ORA     A               ;IF SO, DO SPECIAL DELAY BECAUSE
                                        ;OF DESIGN
                JZ      NTEXDL          ;PROBLEM
                PUSH    H               ;SAVE [H,L]
                LXI     H,19000         ;DELAY COUNT
```

```
        LPTDLY: DCX      H                ;COUNT DOWN
                MOV      A,H
                ORA      L                ;UNTIL ZERO
                JNZ      LPTDLY
                POP      H                ;RESTORE [H,L] REGS
                STA      LPTLST           ;RECORD LINE FEED LAST
        NTEXDL: MVI      A,2               ;SEND A LINE FEED COMMAND
                OUT      2
                XRA      A                ;RETURN WITH 0 &CC'S=0
                RET
        PRINTR: MVI      A,1               ;TELL LPT TO PRINT
                OUT      2                ;STATUS REG
        FINLP2: STA      LPTLST
                DCR      A                ;[A]=0
                STA      LPTPOS           ;RESET LINE PRINTER POSITION
                RET
                .
                .
                .
        LPTCD2: LDA      LPTPOS            ;GET CURRENT LPT PRINT  HEAD POSITION
                ADD      M
                CPI      LPTLEN           ;WILL THIS NUMBER OVERLAP?
                JMP      LINCHK
                .
                .
                .
        LPTCD3: LDA      LPTPOS            ;GET LINE PRINTER POSITION
                                          ;NOTE: COLUMN WIDTH (CLMWID)=
                                          ;14 CHARACTERS
        NLPPOS           EQU     (((LPTLEN/CLMWID)-1)*CLMWID);POSITION BEYOND
                                          ;WHICH THERE ARE
                                          ;NO MORE COMMA FIELDS, SO
                CPI      NLPPOS           ;COMMA JUST DOES A "CRDO"
                JMP      CHKCOM           ;USE TELETYPE CHECK
                .
                .
                .
        IOCHNL: 0                          ;DEPOSIT BOARD TYPE HERE
                0                         ;CHANNEL GETS DEPOSITED HERE.
        IOREST: LXI      H,IOCHNL          ;GRAB POINTER TO IT
                CALL     HELPIO           ;SET UP THE NEW CONSOLE DEVICE
                CALL     STKINI           ;MAKE STACK OK
                JMP      READY            ;AND TYPE "OK" HOPEFULLY ON GOOD CONSOLE
```

To patch the I/O routines, stop the machine after loading
BASIC and insert the patches using the front panel switches
or read in a tape containing the patches. Restart BASIC at
location zero with all sense switches up. This will prevent
BASIC from modifying the I/O routines. In general, these
guidelines should be followed in writing I/O routines:

1. Insert a JMP at TRYOUT to the custom output routine. Be
    sure the PSW that is saved on the stack when the routine
    is entered is preserved. Make sure all registers are
    left unchanged when the routine is exited.

2. Insert a JMP at TRYIN to the custom input routine.
    Return the input character in the A register and do not
    change any of the other registers.   The PSW may be
    changed.

3. To modify ISCNTC insert a CALL to the custom poll
    routine. This routine returns a non-zero condition code
    setting if no character is present, and zero if a
    character is present. The A register and the condition
    codes may be changed.

4. To change the initialization of the 2SIO board, change
    the "ADI 23Q" to "MVI A,XXX" where XXX is the new
    initialization byte.

5. To change the initialization of the 4PIO board, change
    the "MVI A,54Q" to a "MVI A,XXX" where XXX is the new
    initialization byte.

6. To patch in a new line printer driver change the code at
    LPTCOD.   Note that PRINTW is also called by the routine
    which prints a carriage return line feed. The code at
    LPTCD2 and LPTCD3 must be changed if the line printer is
    not 80 characters wide.

7. To recover from an incorrect CONSOLE command, deposit
    the board ADDRESS in IOCHNL, the board type in IOCHNL+1,
    and start the machine at IOCHNL+2.

        Patching Disk BASIC - the PTD program.   After Disk
BASIC is loaded, deposit the desired patches in memory.
Then examine and run PTD at location 54000 octal. After two
or three seconds, the patched version of BASIC will be saved
on disk. The save is complete when the Disk Enable light on
disk drive zero goes out.

        To save a patched version of BASIC on a disk which did
not previously contain release 4.0 Altair BASIC, track 0
must be copied from a 4.0 disk.

        PTD may also be used to save programs other than BASIC
on tracks 0-4 of a diskette by loading the program after
BASIC is loaded and running PTD.   All memory locations
between 0 and 46000 octal will be saved on tracks 0-4 on
diskette zero.

APPENDIX M
USING ALTAIR DISK BASIC
An Example

        The following is a discussion of how to program a
typical   application   in   BASIC.   The example   is the MITS
in-house inventory system which is designed to   run   on   the
following hardware:

        Altair 8800b computer with 32K memory, PROM memory board
            with the Disk PROM Bootstrap loader and a 2SIO serial
            I/O board
        Two disk drives
        24-line Lear-Sigler CRT terminal
        Line printer

        The most important part of the   design   for   an
application is setting up the files.   Files that are
correctly set up will be easy to use and maintain.   Poorly
set up files will be a perpetual headache, causing either an
eventual rewrite or, more likely, abandonment of the system.

        The first listing at the end of the appendix, INVEN,
contains   modules   from   the   main   program in the inventory
system.   INVEN shows how the central file (a random file) in
the system is set up and how it is handled. The INVEN
listing also shows the use of another random file and a
sequential file.   The CALC listing shows how to read
programs as data files. CODE1 is a partial listing of a
program that will be read as a data file.

        The INVEN modules listed were included to show the
following features:

1. program startup initialization and comments about the
    files used by the program (lines 1-35)

2. what the complete program does (lines 60-1000)

3. an example of how to modify records in a random file
    (lines 900-1040)

4. an example of how sequential files are used (lines
    1800-1868 and 2700-2820)

5. one approach to the problem of handling a random file
    that spans more than one disk (lines 2000-2030)

6. three subroutines (lines 300-340, 9000-9020 and
    9200-9220) that are called by the INVEN modules.

    The function FNY (line 6) is used to round dollar
amounts to thousandths of a cent. FNQ (line 7) is used to
round quantities to thousandths and to convert single
precision amounts to double precision.

    INV3 is fielded once in the program initialization, but
INV1 and INV2 are repeatedly fielded by calls to the
subroutine at line 2000. The IF F>255 (line 60) avoids the
possibility that the program can be stopped by an illegal
function call at line 61.

    PUT statements are the very last statements executed in
the Remove from Inventory module, the Add to Inventory
module, etc. This prevents updating one file but not the
other. (This could happen if PUT Z, R1 was at line 1010.)

    Line 2000 sets Z to 1 and R1 to N if the item wanted,
N, is less than 2001. It sets Z to 2 and R1 to N-2000 if
the item wanted is greater than 2000. Line 2020 then sets
the pointers for the variables in the field statement to
point into either the buffer for INV1 or the buffer for
INV2, depending on whether the item wanted is less than 2001
or greater than 2000.

    The CALC listing is a program which determines if there
are enough parts in inventory to meet projected demands.
Line 60 waits while the disk comes up to speed so the
message "ENABLE DISK 1" will not be printed on the terminal.
Lines 100-140 input up to fifty different product codes and
the number of each product to be built. Line 170 opens a
file for each product that contains the parts required for
the product. Lines 220-250 build up a report heading
extracting the product description contained in line 10 of
each file.

    Lines 120-150 accumulate the number of parts required
for each product into the array Q. If more than 32767 of a
part is required, a pointer is set in the array Q and the
number of the part is accumuulated in the array Q!. This
maneuvering is necessary since the system does not have
enough memory to dimension Q as single precision instead of
integer.

The parts lists for a product are programs saved with
the A option. Since they are programs, their maintenance is
very easy. For example, suppose that part 1071 in the 8800b
is too marginal and that from now on part 1173 should be
used instead. With the parts lists disk mounted on drive 0,
the following sequence will update the 8800b file:

```
LOAD "CODE1"
160,1,1173
SAVE "CODE1",0,A
```

The programmer who is cramped for memory will find that
programs can still be documented adequately if comments are
set up as separate files. The memory used for variables
when a program runs can be used for comments if the comments
are merged in when the program is to be listed.
Alternatively, the program could be listed in two or more
parts. Additional memory can be obtained by bringing BASIC
up without optional functions and with no files.

The main inventory program is set up so that a carriage
return typed in responce to any prompt cause the program to
dump the function descriptions on the CRT and to return to
the FUNCTION NUMBER prompt. If the program were to be run
on a printing terminal, instead of a 9600 baud CRT, it would
not be set up to print the descriptions every time the
operator wanted to get back to the FUNCTION NUMBER prompt.
The list of function descriptions might be taped on the wall
next to the terminal instead.

Listing of INVEN

```
1 DEFINT F-N
2 DEFINT R
3 DEFINT Z
5 DEFDBL P
6 DEF FNY#(Q8#)=INT(Q8#*A#+.5#)/A#
7 DEF FNQ#(Q9!)=INT(VAL(STR$(Q9!))*1000#+.5#)/1000#
8 A$=MKD$(0):B$=MKS$(0):A#=100000#
10 DIM Q$(2),P$(2)
11 '

INV1 ON DRIVE 0 HOLDS ITEMS 1-2000
INV2 ON DRIVE 1 HOLDS ITEMS 2001-4000
INV3 ON DRIVE 1 HOLDS SUMS LOGGED IN AND OUT BY DEPARTMENT
12 '
WEKLYRST AND MONTHRST ARE WRITTEN WHILE THE WEEKLY,
MONTHLY ACTIVE ITEMS LISTS ARE PRINTING;
CONTAIN THE ITEM #S THAT NEED TO BE RESET; AND ARE READ BY
THE WEEKLY,MONTHLY RESETS.
14 '
Q$() <=> THREE ON HAND QTY FOR: P$() <=> THREE PRICES
```

```
       [P(0) OLDEST, P(1) NEXT OLDEST, Q(0)<>0 IF Q(1)<>0,
       Q(1)<>0 IF Q(2)<>0]
       D$ <=> DESCRIPTION   LEFT$(D$,3)="$$$" <=> INACTVE ITEM #
       15 '
       I1$ <=> WEEKLY QTY IN
       I2$ <=> MONTHLY QTY IN
       O1$ <=> WEEKLY QTY OUT
       O2$ <=> MONTHLY QTY OUT
       T$ <=> REORDER LEVEL
       DI1$ <=> WEEKLY $ IN
       ID2$ <=> MONTHLY $ IN
       DO1$ <=> WEEKLY $ OUT
       OD2$ <=> MONTHLY $ OUT
       17 '
       DT1$ <=> WEEKLY DEPT $ TAKEN
       DX2$ <=> MONTHLY DEPT $ TAKEN
       DG1$ <=> WEEKLY DEPT $ GIVEN
       DY2$ <=> MONTHLY DEPT $ GIVEN

       20 OPEN "R",#1,"INV1"
       30 OPEN "R",#2,"INV2",1
       32 OPEN "R",#3,"INV3",1
       35 FIELD #3,8 AS DT1$,8 AS DX2$,8 AS DG1$,8 AS DY2$
       60 PRINT:F=0:INPUT"FUNCTION NUMBER";F:IFF>255THEN63
       61 ON F GOTO 210,350,350,1900,600,900,1700,
          2700,2500,2300,2400,1880,2900'
            2   3    4    5    6    7    8    9    10   11   12   13
            14   15   16
       63 PRINT"1 - ENTER NEW ITEM"
       64 PRINT"2 - LIST ITEM ON CRT (SHORT FORM)"
       65 PRINT"3 - LIST ITEM ON CRT (LONG FORM)"
       66 PRINT"4 - PRINT ITEMS ON LINE PRINTER
       67 PRINT"5 - ADD TO INVENTORY"
       68 PRINT"6 - REMOVE FROM INVENTORY"
       69 PRINT"7 - PRINT WEEKLY DEPT DOLLAR RECORD ON LINE PRINTER
       70 PRINT"8 - PRINT WEEKLY ACTIVE ITEMS LIST ON LINE PRINTER
       71 PRINT"9 - WEEKLY RESET
       72 PRINT"10- PRINT MONTHLY DEPT DOLLAR RECORD ON LINE PRINTER
       73 PRINT"11- PRINT MONTHLY ACTIVE ITEMS LIST ON LINE PRINTER
       74 PRINT"12- MONTHLY RESET
       75 PRINT"13- RESET ORDER LEVELS
       76 PRINT"14- PRINT LISTNG OF ITEMS NEEDING TO BE RE-ORDERED
       77 PRINT"15- DELETE OLD ITEM
       78 PRINT"16- ERRORS BACKOUT
       100 GOTO60
       298 '
       *
       SUB - INPUT PART # & GET RECORD
       *
       300 PRINT:PRINT:N=0:INPUT"PART NUMBER";N:IFN<1THENRETURN
       310 IFN>4000THENPRINT:PRINT"''# TOO HIGH''":GOTO 300
       320 GOSUB2000:GETZ,R1
```

```
330 IFLEFT$(D$,3)="$$$"THENPRINT:
    PRINT""'NO INFORMATION ON PART''";N:GOTO300
340 RETURN
890 '
*
F=6 - REMOVE FROM INVENTORY
*
900 GOSUB300:IFN=0GOTO63
920 DN=-1:INPUT"NUMBER OF ITEMS REMOVED FROM INVENTORY";
    DN:IFDN=-1THEN63
950 IFCVS(Q$(0))+CVS(Q$(1))+CVS(Q$(2))<DNTHENPRINT"
    ATTEMPT TO REMOVE MORE THAN ON HAND":PRINT:GOTO63
960 D0=DN:P=0
970 IFD0<CVS(Q$(0))THEN
    P=P+FNQ#(D0)*CVD(P$(0)):LSETQ$(0)=MKS$(CVS(Q$(0))-D0):
    GOTO1000
980 P=P+FNQ#(CVS(Q$(0)))*CVD(P$(0)):D0=D0-CVS(Q$(0)):
    LSETQ$(0)=Q$(1):LSETQ$(1)=Q$(2):LSETQ$(2)=B$:
    LSETP$(0)=P$(1):LSETP$(1)=P$(2):LSETP$(2)=A$:IFD0THEN
    GOTO970
1000 LSETO1$=MKS$(CVS(O1$)+DN):LSETO2$=MKS$(CVS(O2$)+DN):
     LSETDO1$=MKD$(CVD(DO1$)+P):LSETOD2$=MKD$(CVD(OD2$)+P)
1020 GOSUB9200:IFC%=-1GOTO63
1030 LSETDT1$=MKD$(CVD(DT1$)+P):LSETDX2$=MKD$(CVD(DX2$)+P)
1040 PUT3,C%:PUTZ,R1:GOTO900
1790 '
*
F=9 - WEEKLY RESET
*
1800 PRINT"7 - WEEKLY DEPARTMENT RECORD
1802 PRINT"8 - WEEKLY ACTIVE ITEMS
1804 Z$="":INPUT"HAVE THE ABOVE BEEN LISTED FOR TODAY";Z$
1810 IFLEFT$(Z$,1)<>"Y"THENPRINT:PRINT
     "WEEKLY RESET NOT PERFORMED":GOTO63
1843 OPEN"I",4,"WEKLYRST"
1845 IFEOF(4)THENCLOSE4:KILL"WEKLYRST":GOTO1862
1850 INPUT#4,N:IF 1<=NANDN<=4000 THENGOSUB2000:GETZ,R1
     ELSEPRINTN;"OUT OF BOUNDS. RESET ABORTED.":END
1855 LSETI1$=B$:LSETO1$=B$:LSETDI1$=A$:LSETDO1$=A$:PUTZ,R1
1860 GOTO1845
1862 FORI=1TO20
1864 GET3,I:LSETDT1$=A$:LSETDG1$=A$:PUT3,I
1866 NEXT
1868 GOTO60
1999 '
*
SUB - GET Z,R1 FOR N AND FIELD TO INV1,2
*
2000 Z=1-(N>2000):R1=N+(Z=2)*2000
2020 FIELD Z,4 AS Q$(0),4 AS Q$(1),4 AS Q$(2), 8 AS P$(0),
     8 AS P$(1),8 AS P$(2),40 AS D$,4 AS I1$,4 AS I2$,
     4 AS O1$,4 AS O2$,8 AS DI1$,8 AS ID2$,8 AS DO1$,8 AS OD2$
```

```
2030 RETURN
2690 '
*
F=8,11 - WEEKLY,MONTHLY ACTIVE ITEMS LIST
*
2700 N=1:GOSUB2000:GOSUB2855
2703 IFF=8THENOPEN"O",4,"WEKLYRST"ELSEOPEN"O",4,"MONTHRST"
2705 IT#=0:OT#=0:TT#=0
2710 FORI=1TO2000
2720 GETZ,I:IFLEFT$(D$,3)="$$$"THEN2800
2723 Q0=CVS(Q$(0)):Q1=CVS(Q$(1)):Q2=CVS(Q$(2))
2725 IFF=8THENI!=CVS(I1$):O!=CVS(O1$):I#=CVD(DI1$):O#=CVD(DO1$)
     ELSEI!=CVS(I2$):O!=CVS(O2$):I#=CVD(ID2$):O#=CVD(OD2$)
2727 TT#=TT#+CVD(P$(0))*Q0+CVD(P$(1))*Q1+CVD(P$(2))*Q2
2730 IFI!+O!=0THEN2800
2733 PRINT#4,N+I-1
2735 IT#=IT#+I#:OT#=OT#+O#
2740 IFL9>59ANDKK=0THENGOSUB2850
2750 LPRINTUSING"######";99999!+N+I;
2770 LPRINTUSING"##,###,###";I!,O!,Q0+Q1+Q2,Q0+Q1+Q2+O!-I!;
2780 LPRINTUSING"$$,###,###.##";I#,O#
2790 L9=L9+1
2795 KK=KK+1:IFKK=5THENLPRINT:L9=L9+1:KK=0
2800 NEXT
2810 IFN=1THENN=2001:GOSUB2000:GOTO2710
2811 CLOSE4
2813 LPRINT:LPRINTUSING"TOTAL INVENTORY COST =$$##,###,###.##";TT#
2815 REM *GOTO2820 IN F=7,10
2820 LPRINT:LPRINTUSING"TOTAL IN = $$##,###,###.##";IT#
2830 LPRINTUSING"TOTAL OUT =$$##,###,###.##";OT#
2837 LPRINT:LPRINT
2840 GOTO50
2850 FORJ=L9TO66:LPRINT:NEXT
2855 IFF=8THENLPRINT"WEEKLY";:ELSELPRINT"MONTHLY";
2860 LPRINT" ACTIVE ITEMS LIST";:GOSUB9000
2865 LPRINTTAB(39);"STARTED"
2870 LPRINT"ITEM #     QTY-IN    QTY-OUT    ON-HAND    MO-WITH
     DOLLARS-IN   DOLLARS-OUT"
2880 LPRINT:KK=0:L9=6:RETURN
8990 '
*
SUB - PRINT TODAY'S DATE
*
9000 IFTD$=""THENLINEINPUT"TODAY'S DATE ?";TD$:IFTD$=""THEN63
9010 LPRINT" ";TD$
9015 LPRINT
9020 RETURN
9190 '
*
INPUT DEPARTMENT # AND GET TOTALS
*
9200 C%=-1:INPUT"ENTER DEPARTMENT CODE";C%:IFC%=-1THENRETURN
```

```
9210 IF1<=C%ANDC%<=20THENGET3,C%:RETURN
9220 PRINT"INVALID CODE":GOTO9200
```

          Listing of CODE1

```
5   CODE1
10  PARTS LIST FOR: 8800B
20  OCT 30,1976
90  REM THIS IS THE START OF DATA
100 ,11,1042
110 ,3,1134
120 ,4,1040
130 ,1,1020
140 ,1,1021
150 ,1,1024
160 ,1,1071
170 ,1,1074
180 ,1,2105
190 ,24,348
200 ,2,326
```

          Listing of CALC

```
10  CLEAR600
20  DEFINT A-Z
30  DIM CN(49),NU(49),Q(4000),Q!(200)
40  CLOSE:UNLOAD1
50  INPUT"PLACE DISK WITH PARTS LISTS IN DRIVE 1. HIT RETURN";G$
60  FORK!=1TO5000:NEXT:MOUNT1
90  LINEINPUT"TODAY'S MO/DA/YR ";DT$:H$(0)=DT$+" PARTS AVAILABLE FOR:"
95  '
INPUT QUANTITY OF EACH PRODUCT REQUIRED
*****
100 INPUT"CODE NUMBER(0 WHEN FINISHED)";CN(I)
110 IF CN(I)=0 THEN 150
120 IF CN(I)<1 OR 50<CN(I) THEN PRINT"INVALID CODE NUMBER":
    GOTO 100
130 INPUT"NUMBER OF UNITS TO BE MADE";NU(I)
140 I=I+1:IF I<50 THEN 100
145 '
ACCUMULATE QUANTITY OF EACH PART REQUIRED
*****
150 FOR K=0 TO I-1
160 ONERRORGOTO610
170 OPEN"I",#1,"CODE"+MID$(STR$(CN(K)),2),1
180 ONERRORGOTO0
190 LINEINPUT#1,A$:IFA$=""THEN190
200 IFLEFT$(A$,3)="90 "THEN260
210 IFLEFT$(A$,3)<>"10 "THEN190
220 IFKTHENH$(HK)=H$(HK)+","
```

```
230 HH$=STR$(NU(K))+STR$(CN(K))+"=("+MID$(A$,20)+")"
240 IFLEN(HH$)+LEN(H$(HK))>72THENHK=HK+1
250 H$(HK)=H$(HK)+HH$:GOTO190
260 ONERRORGOTO630
270 IFEOF(1)THEN310
280 INPUT #1,A,QN,PN
290 IFQ(PN)<0THENQ!(-Q(PN))=Q!(-Q(PN))+NU(K)*QN
    ELSEQ(PN)=Q(PN)+NU(K)*QN
300 GOTO270
310 ONERRORGOTO0:CLOSE 1:NEXT K
315 '
GET SECOND HALF OF INVENTORY BACK ON LINE
*****
320 CLOSE:UNLOAD1
330 INPUT"
PLACE INVENTORY DISK #1 IN DRIVE 1. HIT RETURN TO START REPORT";G$
340 FORI!=1TO5000:NEXT:MOUNT1
360 OPEN"R",#2,"INV1"
370 FIELD #2,4 AS Q1$,4 AS Q2$,4 AS Q3$,24 AS G$,40 AS D$
375 '
PRINT REPORT
*****
380 GOSUB570
390 FOR I=1 TO 4000
400 IF Q(I)=0 THEN 530
410 QQ!=Q(I):IFQ(I)<0THENQQ!=Q!(-Q(I))
420 IFL9>59ANDKK=0THENGOSUB560
430 L9=L9+1
440 RN=I
450 IFI<2000THEN460ELSERN=RN-2000:IFFLAG=0THEN
    CLOSE2:OPEN"R",#2,"INV2",1:FLAG=1:
    FIELD#2,4 AS Q1$,4 AS Q2$,4 AS Q3$,24 AS G$,40 AS D$
460 GET #2,RN
470 IFLEFT$(D$,3)="$$$"THENLPRINTI+100000!;
    "********* NO INFORMATION ON PART ********";:
    LPRINTUSING"##,######";QQ!:GOTO520
480 QH!=CVS(Q1$)+CVS(Q2$)+CVS(Q3$):QD!=QH!-QQ!
500 LPRINTI+100000!;D$;" ";
510 LPRINT USING "##,######";QQ!;QH!;QD!
520 KK=KK+1:IFKK=5THENKK=0:LPRINT:L9=L9+1
530 NEXTI:CLOSE:END
560 FORK=L9TO66:LPRINT:NEXT
565 '
PRINT PAGE HEADING
*****
570 FORK=0TOHK:LPRINTH$(K):NEXT
580 LPRINT:LPRINTTAB(52);"NEEDED  ON HAND    EXCESS":LPRINT
590 KK=0:L9=5+HK:RETURN
605 '
TRAP ROUTINE: BAD CODE NUMBER
*****
610 IFERR=53THENPRINT:PRINT"NO CODE";MID$(STR$(CN(K)),2);" FILE"
```

```
620 ONERRORGOTO0
625 '
TRAP ROUTINE: ACCUMULATE INTO Q OVERFLOWED
*****
630 IFERR<>6ORERL<>290THENONERRORGOTO0
640 NQ=NQ+1:Q!(NQ)=Q(PN)+NU(K)*QN:Q(PN)=-NQ
670 RESUME270
```

INDEX