# MINI BASIC

MINI BASIC is an extended version of both TINY BASIC, as proposed in People's Computer Company, and TINY BASIC EXTENDED, as created by Dick Whipple and John Arnold. MINI BASIC has been created and implemented by Dr. Mark Yoseloff of The Micro Computer Company.

Briefly, for those familiar with TINY BASIC EXTENDED, MINI BASIC is a superset of TINY BASIC EXTENDED and a subset of BASIC. In addition to all of the features found in TINY BASIC EXTENDED, MINI BASIC allows for an optional LET in assignment statements, for ON...GOTO and ON...GOSUB commands, and for a rather powerful set of character string and substring handling instructions.

## I. SPECIFICATIONS

MINI BASIC, as currently implemented, is designed to accept input using the standard ASCII character set.

The acceptable range of numeric values is -32,768 to +32,767.

Only integer arithmetic is permitted. The symbols +,-,*, and/ are used for addition, subtraction, multiplication and division, respectively. Expressions are evaluated from left to right with precedence of operations observed. Parentheses may be used to change the order of evaluation. For example,

```
                     3 + 7 * 2 + 4 = 21,
while               (3 + 7)*(2 + 4)= 80.
```

MINI BASIC allows for 26 numeric variables, labeled A through Z. Any variable may be subscripted to form either a one- or two-dimensional array. For example,

```
S(3)
```

would refer to the third element in the array S. All arrays must be specified by a Dimension statement. The maximum allowable dimension is 255.

A Character String, or String, is any collection of acceptable ASCII characters. A literal string refers to the collection of characters itself. A string variable refers to a variable whose value may be set to arbitrary literal values.

MINI BASIC permits strings of up to 72 characters. The maximum desired length of a string variable must be specified by a Dimension statement.

MINI BASIC allows for 26 string variables, labeled #A through #Z. In order, for example, to use the variable #W, it is first necessary to set its maximum length:

```
DIM #W(20)
```

specifies that #W will contain at most 20 characters. Any attempt to set #W equal to a string of more than 20 characters will result in the string being truncated to the dimensioned length -- in this case, 20 characters.

MINI BASIC allows for substrings of a string variable to be specified in one of two ways: single subscript will refer to the portion of the string from the value of the subscript to the end of the string; double subscript will refer to the portion of the string between the two subscripts, inclusive. For example, if

```
                          #C = ABCDEFG,

then,                  #C(4) = DEFG
                     #C(3,6) = CDEF
                     #C(2,2) = B
```

For a double subscript, if the second value is less than the first, an error condition will occur. The only exception is that when the second value is exactly one less than the first, a null or empty substring will result. This is very useful for certain loop situations. Examples:

```
              #D(3,1) will produce Error 11

                     #D(3,2) = null string
```

MINI BASIC allows for the LET statement to be used to perform the operation of concatenation of strings. A + is used to denote concatenation. For example, if

```
                          #A = YES
and                       #B = MAYBE

then,               LET #C = #A + " NO OR " + #B

will result in,           #C = YES NO OR MAYBE
```

NOTE: The variable to be assigned (left side of =) cannot be subscripted. The right side of the equation may contain any combination of variable strings, variable substrings, and literals. A string literal must be enclosed by quotation marks whenever it appears. The only exception to this is in the input statement as described in the next section. The quotation marks enclosing a literal are not considered part of the string. For example,

```
                    DIM#A(3)
                    LET#A = "XYZ"
```

is valid, since the length of the string within the quotation marks is three.

MINI BASIC contains three built-in functions:

RN will generate random numbers between 0 and 10,000. It may be used in place of a numeric expression.

TB(exp) is a tab function which may be used in print statements to skip the number of spaces specified by the numeric expression "exp"

LN(#var) is a length function which will return the current length of the string variable #var. It may be used in place of a numeric expression.

MINI BASIC recognizes the following relational operators:

=, <, >, >=, <>, <=

These may be used to test the relative values of either numeric expressions or string expressions. For string expressions, comparison is made character-by-character starting with the left-most character. Comparison is made of the ASCII values of each character. In order for two strings to be equal, they must be identical character-by-character and of the same length. Leading blanks (spaces) are considered, with a space being smaller in value than any other character.

Instructions may be entered in MINI BASIC either with or without line numbers. Instructions entered without line numbers will be executed immediately. Those entered with line numbers will be stored sequentially and will be executed upon a RUN command. There are seven instructions which can be executed only in the immediate mode. These are:

| | |
|---|---|
| NEW | CLR |
| LST | TRE |
| SZE | TWR |
| RUN | |

There are six instructions which may be executed in either mode. These are:

| | |
|---|---|
| DIM | PR |
| IN | DTA |
| LET | IF |

There are eight instructions which can be executed only as numbered program statements. These are:

| | |
|---|---|
| GOTO | ON...GOTO |
| GOSUB | ON...GOSUB |
| RET | FOR |
| END | NXT |

There are three built-in functions. These are:

RN
TB
LN

Every line entered must be terminated by a "Cr" (ASCII $040_8$). Multiple instructions per line are permitted. Instructions must be separated by a "$" (ASCII $044_8$) in such cases.

All of these instructions will be specified in detail in the next section.

Editing capability and error messages will be described in the following section.

## II. DETAILS OF USAGE OF COMMANDS

The seven immediate execution commands are used in the construction, debugging, storing, retrieving, and execution of programs. These commands are NEW, LST, SZE, RUN, TRE, TWR, and CLR. The use of these commands is as follows:

NEW - Start new programming: The NEW command is used to prepare the system for the entry of a new program via the keyboard. The effect of the NEW command is to delete all programming resident in the system and to reset a variety of parameters in preparation for a new program.

LST - List programming: The LST command is used to list all, or a portion of, the resident program. There are three possible formats for the LST command. These are:

| | |
|---|---|
| LST | List entire program |
| LST nnnnn | List line nnnnn only |
| LST nnnnn,mmmmm | List lines nnnnn through mmmmm inclusive |

(Note: Both nnnnn and mmmmm must be existing line numbers)

SZE - Display program size: The SZE command is used to display the amount of storage required by the resident program. Two numbers are displayed; the first is the amount of storage used by the resident program, and the second is the amount of storage still available in the user area of memory.

(NOTE: Prior to the first execution of a new program no space is allocated to arrays and string variables, and, therefore, the first value displayed will not reflect this required space. After the first execution, the first value will increase to reflect this usage. In writing a new program the user should be aware of the fact that an array will require twice the number of array elements plus two bytes and that a string variable will require the dimension of the string plus three bytes once the program is executed.)

TRE - Read from tape: The TRE command is used to read a program from a cassette and place the program in the user area of memory. In practice, to use the TRE command, the user should key-in the letters TRE without a carriage-return. The tape should then be started, and once the leader has begun, the carriage-return should be depressed.

TWR - Write to tape: The TWR command is used to write a program from the user area of memory to a cassette. To use this command, the user should place the cassette unit in the record mode. Once the tape is running the command TWR should be keyed-in followed by a carriage-return.

RUN - Execute resident program: The RUN command is used to initiate execution of the resident program. Execution begins with the lowest line number.

To execute a fully-developed program that is already on cassette the procedure would be:

1. Enter TRE on keyboard (do not type carriage-return)
2. Begin tape
3. Once the leader has been reached, type carriage-return
4. Upon completion of reading-in, type RUN

CLR - Clear screen: The CLR command will clear the screen without disturbing the resident program.

The remainder of this section is devoted to descriptions of the commands which may be executed either as immediate commands or stored with line numbers for later execution, and those which may be executed only as part of a resident program.

The format for entering a line of coding for inclusion in the resident program area is as follows:

nnnnn (statement)cr

where: nnnnn is the line number of the statement. All statements are stored sequentially in ascending order of line number.

$0 < nnnnn < 65535$

(statement) is an optional statement consisting of either a single command and its associated operands or several such commands separated by $.

cr is a carriage-return (ASCII $015_8$)

NOTES:

1. Whenever a line is entered which has a line number that is a duplicate of an existing line number, one of the following will occur:

   a. If the new line contains only a line number immediately followed by a carriage-return, the old line is deleted and the current line is NOT included in the program. The net effect is to delete the line from the program.

   b. If the new line contains anything except a line number immediately followed by a carriage-return, it replaces the old line.

   c. The line-delete function CTL-L (described in the next section) will not have the same effect as (a) above. CTL-L will clear the input buffer but will not delete anything already resident in the system. CTL-L will only delete those characters not already accepted by the system via the typing of a carriage-return.

2. A line of coding can be inserted in the resident program between two already resident lines by simply giving it a line number which falls between those of the already resident lines. The system will automatically insert it in the proper position.

3. In entering lines of coding a space MUST BE USED between the last digit of the line number and the first character of the statement. The only exception is that a space MUST NOT be used in deleting a line (see 1.a. above).

4. No spaces are necessary within the statement itself except for one necessary space in certain uses of the IF command, and the ON...GOTO and ON...GOSUB commands. See below, under these commands, for the specific exceptions.

In describing the MINI BASIC commands, the following notation will be used;

| | |
|---|---|
| var = | any numeric variable, whether subscripted or not, unless otherwise specified |
| strvar = | any string variable, whether subscripted or not, unless otherwise specified |
| avar = | any variable as described under var and strvar above |
| expr = | any valid numeric expression, which may involve numbers, and variables, whether subscripted or not |
| strlit = | any string literal |
| strexpr = | any string variable, whether subscripted or not, or any string literal |
| nnnnn = | line number |
| [ ] = | indicates optional portion of statement |
| ƀ = | blank (space) |

1. DIM - Assign array or string dimensions: This command is used to allocate space in memory for the storage of arrays or string variables. Numeric arrays may have one or two dimensions. Every string variable used MUST BE DIMENSIONED, even if the dimension is 1. The DIM statement takes one of the following forms:

```
[nnnnn] DIM var(expr)
[nnnnn] DIM var(expr-1, expr-2)
[nnnnn] DIM strvar(expr)
```

NOTES:

1. var and strvar may not be subscripted.

2. Several variables may be dimensioned by a single DIM statement, with the separate elements separated by commas.

Examples:

```
 20 DIM X(40)
100 DIM D(5,X+7)
 70 DIM #F(10), B(4,7), #C(1)
```

2. IN - Enter (input) from the keyboard: The IN command will cause the system to display

?ƀ

on the screen requesting data to be entered. More than one item may be specified by an IN command. The format of the IN statement is

```
[nnnnn] IN avar-1, avar-2, ..., avar-n
```

NOTES:

1. If more than one variable is specified in the IN statement, the values are entered one at a time and a carriage-return is required between each value entered.

2. A string variable specified in an IN statement may not be subscripted.

3. A string variable input will cause the variable to be set equal to the literal that is entered. The literal may be entered with or without enclosing quotation marks. If quotation marks are used they will be ignored in setting the value of the variable. This exception to the strict grammar rule requiring enclosure of literals in quotes facilitates string entry by those totally unfamiliar with the MINI BASIC language.

4. A semi-colon can be used in place of a comma to separate the variables in a multivariable IN statement. The comma will result in a carriage-return and line feed between each entry. The semi-colon will inhibit the crlf, permitting the data to be entered on a single line.

   Examples:

```
 25 INA, #C
200 IN D(2,3)
 30 IN #B; #C; X, B(1)
```

3. LET - Assign a value to a variable: The LET statement permits the assignment of a value to a variable. The word LET is optional in assignment statements and may be omitted. The LET statement may be used to concatenate strings. Concatenation is indicated by the use of the symbol +. The LET statement may take any of the following forms:

```
   [nnnnn] LET var = expr
       [nnnnn] var = expr
[nnnnn] LET strvar = strexpr-1 + strexpr-2 + ... + strexpr-n
    [nnnnn] strvar = strexpr-1 + strexpr-2 + ... + strexpr-n
```

NOTES:

1. The assignment statement in which the word LET is omitted will require several hundred microseconds longer to execute than when the word LET is included. This will be meaningful only in the most extreme cases, where the assignment statement is contained in a loop that will be executed a large number of times. Generally, this delay should be ignored in favor of the space saved in the user area of memory by omitting the word LET.

2. In the assignment of string variables, the variable being assigned (left-hand side of equality may not be subscripted).

   Examples:

```
      25 LET A = 7
100 LET H(2,3) = (A(1,4) + B(7))/25
      200 X(4) = 25 + (100+B)/3
     30 LET #C = #B(2,3)
        550 #W = #A(2,4) + "XYZ" + #C(7)
```

```
100 LET #A = #A(1,J-1) + " - " + #A (J+1)
     50 #Q = #Z
```

4. PR - Print output data: The PR command is used to display data on the screen. Several items may be specified in a single PR statement. The individual items must be separated by either a comma or a semi-colon.

The PR statement was designed, initially, to be used with a teletype machine and, therefore, an output line is assumed to be 75 characters long. When print items are separated by commas, the print-line is assumed to be divided into five 15 character fields; each item is left justified in the next sequential print field. When individual items are separated by a semi-colon a single space is inserted between the items as they are printed. Care should be taken in using the comma as a separator, since the screen width is 32 characters. The first two data fields will print on one line (the first 30 characters) but the third field will be split between two lines.

A PR statement can be terminated in any of three ways. The first method is to follow the last item with a carriage-return. This will cause the screen to scroll up one line after printing. The second method is to use a comma after the last item and then a carriage-return. The third method is to use a semi-colon after the last item and then a carriage-return. The comma will advance the print field pointer to the next 15 character print field and the semi-colon will advance the print field pointer one space. Neither of these will result in the screen scrolling unless the advancing of the print field pointer causes the pointer to go past the 32nd character on the line. In that case, the screen will scroll.

The format of the PR statement is:

```
[nnnnn] PR [print-item] [separator print-item] ...
                        [separator print-item] [separator] cr
```

Here, the separator is either a comma or a semi-colon, and the print-items are any string or numeric expressions.

NOTES:

1. A statement of the form [nnnnn] PR cr will cause a carriage-return and line feed. The effect will be to scroll the screen up one line.

2. In the immediate mode the PR statement can be used to cause the system to act like a calculator. For example,

   ```
   PR ((16*(27+4)+8*(2+5))/3)*(8+18)
   ```

   will result in 4784 being printed on the next line.

3. The function TB is used exclusively in PR statements. The format is

   TB(expr)

   where $0 \leq expr \leq 255$

   The TB function can replace any print item and will cause as many spaces as the value of expr to be printed out. If expr = 0, 256 spaces will be outputted, clearing half of the screen.

Examples:

```
 20 PR "THE VALUE IS"; X
 50 PR TB (5); "YOUR NAME IS"; #N
100 PR A(2); "IS YOUR NUMBER"
120 PR "ENTER YOUR SELECTION";
200 PR "THE LETTER IS"; #W (3,3)
```

5. DTA - Assign value(s) to numeric variables: The data statement is used to assign values to individual variables or consecutive variables. The format is:

```
[nnnnn] DTA var-1 = expr-11[,expr-12,...,expr-ln]
                    [;var-2 = expr-21, ..., expr-2m]
                    ...[;var-k = expr-kl, ..., expr-kp]
```

In the following discussion, reference will be made to consecutive variables. For simple (non array) variables this will mean consecutive letters of the alphabet. For arrays this will mean consecutive elements of the array. For a two-dimensional array A, of dimension m x n, the ordering is:

A(1,1), A(2,1), ..., A(m,1), A(1,2),
..., A(m,2), ... A(1,n), ..., A(m,n)

So, for example, if DIM A (3,4), the following diagram would apply:

$$\begin{tikzcd}
A(1,1) \arrow[r] & A(2,1) \arrow[r] & A(3,1) \arrow[dll] \\
A(1,2) \arrow[r] & A(2,2) \arrow[r] & A(3,2) \arrow[dll] \\
A(1,3) \arrow[r] & A(2,3) \arrow[r] & A(3,3) \arrow[dll] \\
A(1,4) \arrow[r] & A(2,4) \arrow[r] & A(3,4)
\end{tikzcd}$$

To use the DTA statement to assign individual values, the entries are separated by a semi-colon. For example,

```
300 DTA C=7;R(2,3)=21;X=-2
```

would result in setting the values of C, R(2,3) and X, as indicated.

The DTA statement can also be used to assign values to consecutive variables. In such a case, the values to be assigned are separated by commas. For example, the statement

```
20 DTA D=3,7,-4
```

would result in setting

D = 3
E = 7
and F = -4

As further examples,

```
40 DIM C(2,3)
50 DTA A=50,2;C(1,1)=5,4,3,2,1;X=7
```

sould result in setting

```
     A=50
     B=2
C(1,1)=5
C(2,1)=4
C(1,2)=3
C(2,2)=2
C(1,3)=1
     X=7
```

NOTE: EXTREME CARE must be taken in using the DTA statement to assign values to consecutive variables. In the above example, if the statement read

```
50 DTA A=50,2,6;C(1,1)=5,4,3,2,1;X=7
```

the effect would be to set

```
A=50
B=2
```

then, C would be set to 6 and the array dimensioned in 40 would be wiped out.

As another example,

```
20 DIM B(8)
30 DTA B(3)=1,2,3,4,5,6
```

will set

```
B(3)=1
B(4)=2
B(5)=3
B(6)=4
B(7)=5
B(8)=6
```

If statement 30 read

```
30 DTA B(3)=1,2,3,4,5,6,7,8
```

the effect of this statement would be to set B(3) through B(8) as above. The system would then attempt to set values for B(9) and B(10), which will result in the destruction of another array in memory.

The following rules must be observed: IN ASSIGNING CONSECUTIVE SIMPLE VARIABLES AN ASSIGNMENT MUST NOT SET A VALUE FOR A DIMENSIONED ARRAY. IN ASSIGNING VALUES TO CONSECUTIVE ELEMENTS OF AN ARRAY THE DIMENSION OF THE ARRAY MUST NOT BE EXCEEDED.

Failure to observe these rules will not result in an error message being transmitted. Execution will continue with incorrect values for the variables in question.

6. IF - Conditional statement: The IF statement is used to test the comparative size of two expressions and alter the program execution based on the result. The IF statement can have one of the following formats:

```
[nnnnn] IF expr-1 relop expr-2 ƀ imperative statement
[nnnnn] IF strvar relop strexpr ƀ imperative statement
```

where: relop is any of the six relational operators listed in the previous section and the imperative statement is any valid MINI BASIC statement. If the imperative statement is another IF statement the effect is that the logical AND of both IF statements is implied. That is, both must be true for the statement to be true.

The imperative statement will be executed if the condition is true. Several imperative statements may be combined using a $ as a separator. If the condition is false execution will be passed to the next instruction in the program.

NOTE: The format of the IF statement requires a blank (space) between the condition portion of the statement and the imperative statement. Failure to insert this blank will result in an Error 12 being issued in certain cases. In particular, this will be the case if the second expression in the condition portion of the statement is a simple (nonsubscripted) variable. In other cases the blank may be omitted. For example,

```
     20 IF A=27PR"OKAY"
and 100 IF A(2,3)=(D-4)GOTO250
```

will execute properly. However,

```
25 IFB(3,4)=XGOTO250
```

will result in an error. In this case the statement must read:

```
25 IFB(3,4)=X GOTO250
```

Examples:

```
100 IF(X+2*Z)>(3*A) A=5$GOTO1000
 50 IF#A(1,3)="YES"GOSUB30
 70 IF#A(2,2)=#B(1,1)IFZ=3IF#W<#CƀPR"TRY AGAIN"$GOTO20
```

The blank in the last example is required because #C is a simple variable.

The commands which can be executed only as part of a stored program will now be examined.

1. GOTO - Alter program execution sequence: The GOTO command is used to jump to a statement other than the next consecutively numbered statement in executing a program. The format of the GOTO command is:

```
nnnnn GOTO expr
```

expr, when resolved into a numeric value, must equal an existing line number in the resident program. If expr is anything other than a numeric value, the effect is that of a computed GOTO.

Examples:

```
250 GOTO 100
 50 GOTO (R(3)+X)
```

2. GOSUB - Jump to a subroutine and save address of next sequential line number for return using the RET statement: The GOSUB command will result in program execution transferring to the line number specified in the statement. Execution will continue from that point, sequentially, until a RET command is encountered. At that point execution will transfer back to the statement immediately following the GOSUB command. The format of the GOSUB command is:

```
nnnnn GOSUB expr
```

expr, when resolved into a numeric value, must equal an existing line number in the resident program. If expr is anything other than a numeric value, the effect is that of a computed GOSUB.

Examples:

```
 50 GOSUB 200
170 GOSUB 250+3*X
```

3. RET - Return from subroutine: The RET command is used to terminate a subroutine. The effect is to transfer execution back to the statement immediately following the GOSUB command which transferred execution to the subroutine.

4. ON . . . GOTO - Jump to one of a number of portions of the resident program based on the value of an expression: The ON . . . GOTO statement will cause execution to jump to one of a number of specified line numbers, based on the value of a given expression. The format of the ON .... GOTO command is:

```
nnnnn ON expr ƀ GOTO expr-0, expr-1, expr-2,..., expr-n
```

The ON . . . GOTO command will cause the execution to jump to expr-0 if expr is less than or equal to 0; execution will jump to expr-i for values of expr between 1 and n. For values of expr greater than n, execution will jump to expr-n. The destination value, expr-i, must resolve into a line number in the resident program. The blank between expr and GOTO is required in certain uses of the ON . . . GOTO. (See the note under the IF command for a discussion of the required blank.)

Examples:

```
30 ON X GOTO 250,600,420,380,700
```

In this case, if $X \leq 0$, then execution will transfer to 250. If X=1, transfer will be to 600; for X=2, transfer will be to 420; for X=3, transfer will be to 380; for $X \geq 4$, transfer will be to 700.

As another example:

```
100 ON (2*S+(P-Q)/T) GOTO D(1,1),D(2,1),R,250,45
```

5. ON . . . GOSUB - Jump to one of several subroutines based on the value of an expression: The ON . . . GOSUB is identical in operation to the ON. . . GOTO (see above) except that a RET command in the subroutine will transfer execution back to the statement immediately following the ON . . . GOSUB statement. The format is:

```
nnnnn ON expr ␢ GOSUB expr-0, expr-1, expr-2, ..., expr-n
```

6. FOR - Set up a program loop: The FOR command in conjunction with the NXT command (see below) is used to set up program loops. The format is:

```
nnnnn FOR var = expr-1 TO expr-2
```

The FOR command establishes the starting and ending values of the loop control variable, var. Execution of the FOR command fixes these values for the duration of the execution of the loop- If the values of expr-1 and expr-2 are altered during the execution of the loop this will not alter the starting or ending values for the execution of the loop. The end of the loop is indicated by the use of a NXT command.

Examples:

```
100 FOR I = 1 TO 10
    - - -
    - - - statements
    - - -
200 NXT I


 50 FOR F(2) = 1 TO Y-17
    - - -
    - - -
    - - -
 80 NXT F(2)
```

The increment used in advancing the loop control variable is always 1. Care should be taken in exiting a loop other than by incrementing through all values of the loop control variable, since the loop will remain open until terminated in this way.

7. NXT - End of loop: The NXT command is used in conjunction with the FOR command (see above). The NXT command causes the loop control variable to be incremented by 1. This value is then tested against the value of expr-2 in the FOR command. If this value is less than or equal to expr-2, execution is transferred to the statement immediately following the FOR command. If the terminal condition is met then control is transferred to the statement immediately following the NXT command.

The format of the NXT command is:

```
nnnnn NXT var
```

NOTE: FOR - NXT loops may be nested. Care should be taken in observing the order of the loop control variables in such nests.

Example:

```
100 FOR I = 1 TO 50
    - - -
    - - -
150 FOR J = A TO F(3)
    - - -
    - - -
175 FOR K = 20 TO 40
    - - -
    - - -
    - - -
220 NXT K
    - - -
240 NXT J
    - - -
    - - -
300 NXT I
```

8. END - End of program: The END statement is used to indicate the end of the resident program. Execution of this statement will cause the system to exit from the resident program and enter the immediate mode. The END statement should generally appear as the last statement in numerical sequence. The format is:

```
nnnnn END
```

The remainder of this section is devoted to a description of the built-in functions resident in MINI BASIC.

1. TB - Tab function: This function is used exclusively with the PR statement. See the description of the PR statement for a description of its use.

2. RN - Random number generator: The function RN will return a random number in the range $0 \leq RN \leq 10000$. RN can be used wherever a variable occurs in an expression.

3. LN - Length of a string: LN returns the length of a string variable. The format for using LN is LN(strvar). In this case, strvar cannot be subscripted. The value returned by LN is the actual current length of the string variable, not its dimension. For example, if DIM#V(30), and #V="ABCD", then LN(#V)=4. LN may be used wherever a variable occurs in an expression.

## III. EDITING CAPABILITY, ERROR MESSAGES, AND GENERAL USE

Loading of the MINI BASIC cassette will cause the system to display the words "MINI BASIC" followed by two colons:

```
MINI BASIC
:
:
```

At this point, any of the MINI BASIC immediate commands can be entered. For example, a previously recorded program may be read in by keying in TRE and a Carriage Return as described in the previous section. Once the reading is completed, the screen will again clear and the message "MINI BASIC" followed by a single colon will appear. Keying in RUN will then cause the program to be executed.

Whenever the system is halted in the immediate command mode a colon will be displayed. During program execution, the command mode can be entered in one of two ways:

1. If the system is halted during execution awaiting data input, then the immediate command mode can be entered by keying an ESCape (ASCII $033_8$). This will halt execution and produce a colon on the screen.

2. At any time, a system (hardware) reset/restart will cause the execution to halt and the system will enter the immediate command mode.

During the creation and debugging of new programs, MINI BASIC allows program editing in several ways. The previous section describes methods for deleting and replacing lines of MINI BASIC code which are already resident in the system. (Recall that a line becomes resident in the system by keying a "Cr" following the entry of the line.)

Prior to keying a Carriage Return, the newly entered line resides in the input buffer. The line currently in the input buffer may be edited in one of two ways:

1. If one or a few characters are to be deleted this can be accomplished by keying one DELete (ASCII $177_8$) character for each character to be deleted. Each time the DELete is keyed the symbol "←" will appear on the screen and the last entered character in the input buffer will be deleted (the input buffer pointer will be decremented by one). For example, if

   ```
   100 X = A + B
   ```

   has been keyed in, this may be changed to

   ```
   100 X = A - B
   ```

   by keying DELete twice followed by

   ```
   - B
   ```

   The appearance on the screen would be

   ```
   100 X = A + B← ← - B
   ```

2. If an entire line is to be deleted before a carriage return has been keyed, this can be accomplished by keying a CTL-L (Control L, ASCII $014_8$). This will clear the input buffer entirely.

MINI BASIC has the capability to detect a number of program and data errors. Detection of an error will cause an error message to be displayed, followed by a return to the immediate command mode. The form of error messages is:

ERR ee nnnnn

where: ee is the error number

nnnnn is the number of the line where the error has been detected

There are 16 error messages generated by MINI BASIC. These are:

1. Input line too long: This error occurs when there is an input buffer pointer overflow. That is, more than 72 characters have been entered.

2. Input numeric overflow: This error occurs when numeric input is called for and the input value is outside of the range -32768 to +32767.

3. Illegal character: This error occurs during execution when an unrecognizable character appears in a resident statement.

4. Missing quotation mark: This error occurs when a string literal is not properly enclosed by opening and closing quotation marks.

5. Arithmetic expression too complex: This error will require a complex arithmetic expression to be divided into several parts and re-entered.

6. Illegal arithmetic expression.

7. Label does not exist: This error occurs when a jump is specified and the destination line number does not occur in the program.

8. Division by zero: This error may require a test for zero to be inserted in the program before division is attempted.

9. Subroutines nested too deep: This error occurs if more than eight subroutine calls are nested (occur before a return).

10. RET with no GOSUB: This error occurs when the program contains a RET statement which occurs without a subroutine call.

11. Illegal variable: This error occurs when a character other than the characters A through Z or #A through #Z appears as a variable in a MINI BASIC statement.

12. Unrecognizable statement: This error occurs when a command other than those permitted by MINI BASIC appears.

13. Parentheses error: This error occurs when the number of left and right parentheses are unequal in an arithmetic expression.

14. Memory depletion I: This error occurs when the resident program memory area has been exhausted.

15. Memory depletion II: This error occurs when the resident program does not exceed the user memory area but the resident program plus the dimensioned variable storage does.

16. String too long: This error occurs when the concatenation of several strings produces a string longer than 72 characters.

A final interesting point about MINI BASIC is that several MINI BASIC programs can share the same set of variable values. This is particularly useful in small systems with limited storage size. To understand this point, the user should be aware that reading in a new program does not destroy the values of variables already set, unless the new program extends into the dimensioned variable storage area.

In any case, to use this technique, the user need only be certain that the programs which are to share variable values contain identical DIM statements for the dimensioned variables to be used in common, and that these DIM statements occur before any other DIM statements in any of the programs.

As a very simple example to demonstrate this point, consider the following three programs:

Program 1:

```
10 DIM#A(15),#B(30),A(2),F(2)
20 #A = "THIS IS A TEST"
30 #B = "A(1)+A(2)+B+C+D+E+F(1)+F(2)="
40 DTA A(1) = 3,7;B=-3,8,27,-18;F(1)=-8,20
50 PR"DONE"
60 END
```

Program 2:

```
10 DIM#A(15),#B(30),A(2),F(2)
20 PR#A
30 PR A(1);A(2);B;C;D;E;F(1);F(2)
40 END
```

Program 3:

```
10 DIM#A(15),#B(30),A(2),F(2)
20 X=A(1)+A(2)+B+C+D+E+F(1)+F(2)
30 PR#B;X
40 END
```

If Program 1 were loaded, then Program 2, then Program 3, the variable values would pass from one to the other. After running Program 1, the screen would display

```
DONE
:
```

After running Program 2, the screen would display

```
THIS IS A TEST
3 7 -3 8 27 -18 -8 20
:
```

After running Program 3, the screen would display

```
A(1)+A(2)+B+C+D+E+F(1)+F(2)=36
:
```

Of course, in this simple example the use of the technique is unnecessary. However, this technique can be used to divide a large program which would otherwise not fit into several sections which will share the same variable values. Since each program portion must be loaded from tape prior to execution, it is best to divide the large program in such a manner that each program section (except the last to be loaded) need not be executed again.