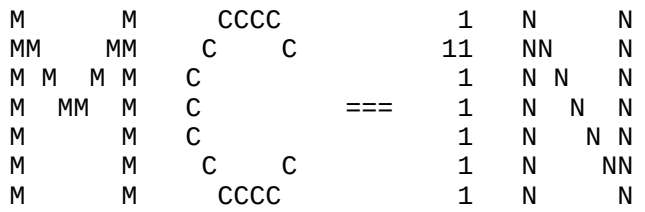
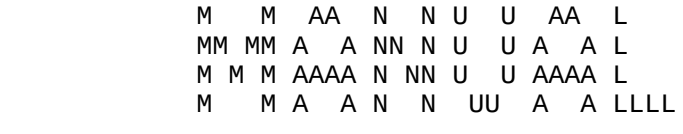

M M CCCC 1 N N
MM MM C C 11 NN N
M M M M C 1 N N N
M MM M C === 1 N N N
M M C 1 N N N
M M C C 1 N NN
M M CCCC 1 N N


M M AA N N U U AA L
MM MM A A NN N U U A A L
M M M AAAA N NN U U AAAA L
M M A A N N UU A A LLLL

==========
==========
==========
========== BASICON
==========
==========
==========

503-626-1012

NSC is a trademark of NATIONAL SEMICONDUCTOR CORP.

Preface

Your BASICON Microcontroller (MC-1N) is a fully self-contained general purpose programmable controller with CPU, RAM, ROM, Real Time Clock, I/O, Communications Circuitry and "INS 8073 Tiny BASIC" on a 3 inch by 4 inch board. Just apply +5Vdc and connect to a terminal and you are up and running.

This document is organized in three chapters and an appendix:

1. Unpacking and Setup.

2. Language Specification.

3. Product Description.

4. Appendix.

CHAPTER 1

UNPACKING AND SETUP

1.1 UNPACKING YOUR MC-1N

Carefully unpack your order comparing the contents against the enclosed packing list. If there are any discrepancies notify BASICON promptly

1.2 CONNECTIONS TO MC-1N

There are two connectors on the MC-1N, J1 (10 pins) and J2 (26 pins). J1 is the power and communications connector. J2 is the Input/Output connector and will be discussed in detail in chapter three. See section 3.5 page 66 for more information.

1.2.1 Applying Power

Connect a +5Vdc +/- 5% power source capable of delivering at least 200 mA to pin 10 of J1. Connect the 5 volt return (or ground) to pin 9 of J1.

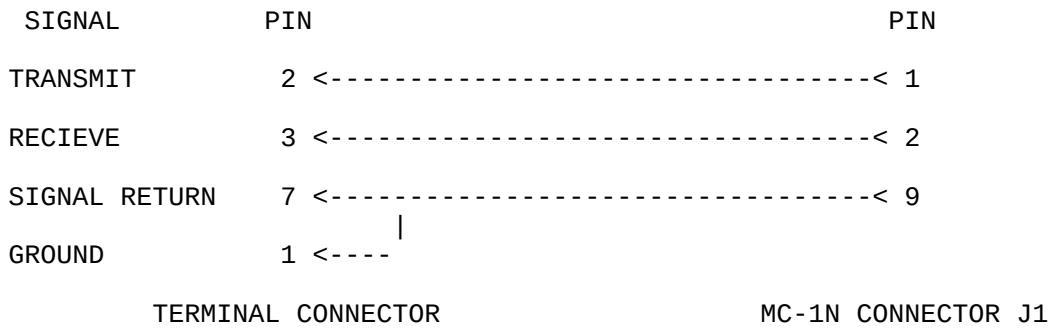
1.2.2 The terminal

The MC-1N will connect to a wide variety of terminals through its RS-232 signal lines. A cable will need to be prepared to meet the connector and strap options of your particular terminal. See section 3.5 page 66 for more information.

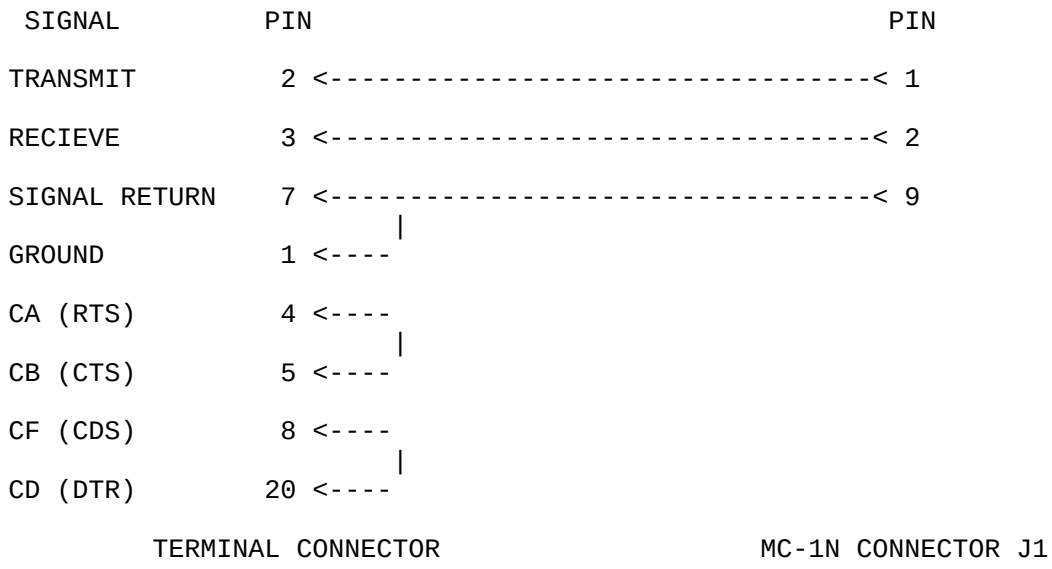
The RS-232 signals are present on J1 as follows:

- Pin 1 is serial input to the MC-1N.
- Pin 2 is the serial output from the MC-1N.
- Pin 9 is the signal ground (system ground).

For a simple terminal with RS-232 inputs only and no modem control lines, the terminal cable would be as follows:

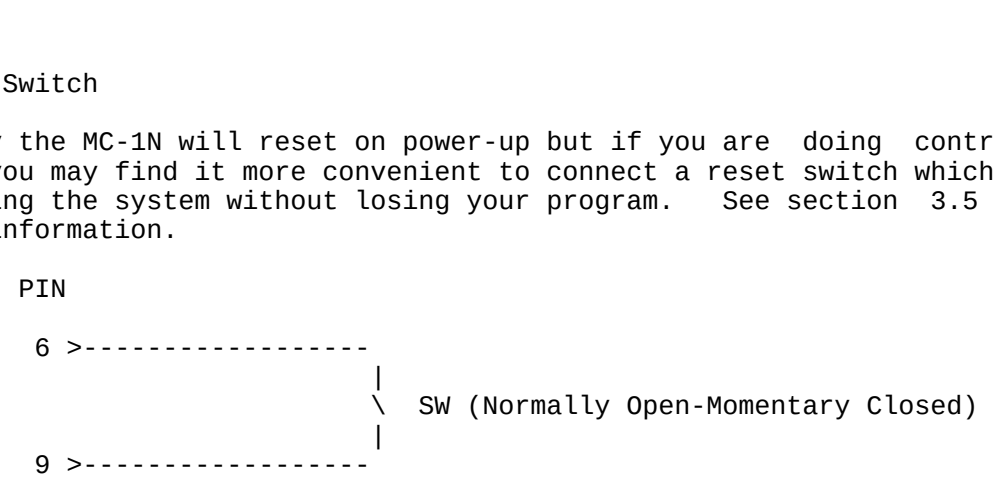


Due to industry standards not being universally applied the same by all manufacturers, you may find the subminiature 'D' connectors on the back of your terminal to be either male or female types. In addition, many terminals need modem control signals applied to enable receiving or transmitting to occur. See section 2.5 page 66 for more information. Therefore a cable to meet the above requirements might look as follows:



1.2.3 Reset Switch

Normally the MC-1N will reset on power-up but if you are doing controller development you may find it more convenient to connect a reset switch which will allow resetting the system without losing your program. See section 3.5 page 66 for more information.



The system reset line is present on:

- Pin 6 - Reset line (normally high)
- Pin 9 - ground

You've made connections to your terminal now your terminal must be configured to communicate with the MC-1N.

- RS-232 (or TTL) - Selected (current loop not supported)
- Data Bits - 8
- Parity - None
- Stop Bits - 1 or 2 (MC-1N sends 1)
- Duplex - Full
- Baud Rate - 110, 300, 1200 or 4800

The MC-1N can communicate at four different baud rates as indicated above. Once you've chosen a baud rate set both the terminal and the MC-1N to the same value.

The baud rate on the MC-1N is selected by jumpers E4 and E5 as follows:

BAUD RATE	E4	E5
110	OUT	OUT
300	OUT	IN
1200	IN	OUT
*4800	IN	IN

* We have found that 4800 baud may cause stray characters to appear on some terminals. Therefore, use it only after you are confident that your systems operation is proper.

1.3 LET'S FIRE IT UP

After checking that all connections have been properly made, as described above, apply power to the MC-1N. When the terminal has warmed up the symbol ">" should appear in the first character position of the terminal screen. If a character (or characters) other than a ">" appear the baud rates of the terminal and MC-1N may not be the same. If the MC-1N is radiating excessive heat or making noise or smoke there is a serious problem which will require attention before the system will function properly. If nothing is happening recheck all connections and if you are still having problems feel free to contact BASICON for assistance.

If a ">" appears on your terminal, press the return (or enter) key. Each time the return key is pressed another ">" will appear below the previous one. If the initial ">" appears each time the power is applied or the reset switch is pressed, but a new ">" does not appear when the return key on the terminal is pressed then check your wiring connections. The terminal is receiving data at the correct baud rate, but the MC-1N is not receiving data from the terminal.

If all has gone well to this point then it is time to try some BASIC programming. The first thing to do is enter the command NEW #1100. This will put the MC-1N BASIC program memory pointer at the first available RAM memory location. The next command is to type NEW. This will set the top of memory pointer (TOP) to the next available RAM memory location, which at this point is address #1101. Now the MC-1N is ready to be programmed. The BASIC interpreter can be given keywords and will respond to them in the "IMMEDIATE" mode. By prefixing each line with a line number the lines will be stored in the program memory space. Lines may be entered in any order because Tiny BASIC will insert them in numeric order in program memory.

GOOD LUCK and GOOD PROGRAMMING

1.4 HOW TO GET HELP

FOR INFORMATION OR IMMEDIATE SERVICE, CALL 503-626-1012
BASICON, INC, 11895 N.W. Cornell Rd., Portland, Oregon 97220

CHAPTER 2

LANGUAGE SPECIFICATION

2.1 GENERAL INFORMATION

2.1.1 Design Considerations

The original language of BASIC developed at Dartmouth College is designed for people who have had no previous experience with computers. Because NSC Tiny ASIC is a descendant of Dartmouth BASIC, it has similar syntax and is easy to learn and use. However, NSC Tiny BASIC is designed specifically for process control. Some Dartmouth BASIC features which are inappropriate to INS 8073 applications have been left out of NSC Tiny BASIC. Among these are trigonometric and other transcendental functions, array and fractional numbers. To further conserve memory space, all redundant commands and statement types which can be duplicated by combining other commands have also been eliminated.

However, NSC Tiny BASIC allows fast hardware tests, examination and modification of any memory location or input/output port, bit by bit examinations of any port, bit manipulation, and logical operations. The NSC Tiny BASIC interpreter can process both decimal and hexadecimal values. An NSC Tiny BASIC program may also access machine language code as a subroutine.

Once the application program has been developed and tested, the MC-1N may be converted from program development to automatic program execution mode. When the development program is stored in an EPROM located in address #8000, the NSC Tiny BASIC interpreter will execute it every time the system is powered up or reset.

2.1.2 NSC Tiny BASIC's Execution Modes

NSC Tiny BASIC executes commands in one of two modes:

Run
Immediate

The system is ready to accept a command when the NSC Tiny BASIC prompt, a right pointing arrow head ">", appears at the left edge on a new line at the terminal. To give an instruction in the immediate mode, enter a command keyword, for example PRINT"HI". The command is executed when the carriage return key is pressed. The above command will print HI on the next line of the terminal The prompt will appear on the line following HI.

Programs are edited and interactively debugged in the immediate mode. Some NSC Tiny BASIC commands, such as RUN, LIST, CONT, and NEW, are used exclusively in the immediate mode. Others, such as GOTO and LET, are used in both modes. Still others, such as INPUT and GOSUB cannot be used in the immediate mode.

To enter the run mode, enter the command RUN in the immediate mode. If there is a program in memory, it is executed. The system will return to the immediate mode when program execution is complete, interrupted by an error, BREAK, or control C key.

2.1.3 Program Line Syntax

Unlike Human Beings, all computer languages operate under very strict grammaric rules. In the computer world these grammar rules are called syntax.

As with any computer language NSC Tiny BASIC requires strict adherence to its syntax rules. Therefore, when you first begin programming you will probably experience more syntax errors than anything else. As you become more familiar with NSC Tiny BASIC, these syntax errors will decrease.

The syntax is so strict that you must spell all keywords and commands exactly as specified and all statements must be in exactly the correct order for your program to run without errors. However, correct syntax does not guarantee that your program will perform as you intend it to.

Luckily NSC Tiny BASIC has only a handful of commands and statements to learn so you should become proficient in a short time.

A program is a series of instructions which, when executed sequentially by the computer, accomplishes a specific task. It is entered into memory one line at a time. This section describes the elements of a program line as the computer reads them from left to right. A program line consists of a line number and a command statement, as shown below:

EXAMPLE

```
100 PRINT "HELLO"
```

The line number indicates that this instruction is part of a program and should not be executed immediately, so NSC Tiny BASIC stores the line in memory. Line numbers also indicate the sequence in which the instructions are to be executed. Therefore, if other lines are already stored in memory, NSC Tiny BASIC inserts the new line in its numerical place among them. Only values in the range 0 (zero) to 32767 are accepted as valid line numbers.

Several statements may follow a single line number if they are separated by colons. Packing several commands on one line conserves memory space. The number of commands in the line is not limited, but the line may not contain more than 72 characters.

The command statement has two parts: the command keyword and the argument. In the example line above, PRINT is the command keyword and "HELLO" is the argument.

NSC Tiny BASIC recognizes 18 statement keywords. Each specifies a statement type which performs one of three actions:

1. Assignment to a variable (LET)
2. Input or output (INPUT, PRINT)
3. Control flow (IF, GOTO, GOSUB, RETURN, DO, ON, FOR. LINK, UNTIL, STEP, NEXT, STOP, DELAY, and REM)

In the sample program line above, a space separates the keyword PRINT from the argument "HELLO". Although it makes the statement easier to read, the space is unnecessary. Within the statement portion of a program line, NSC Tiny BASIC ignores all spaces. Any spaces entered remain in the program and take up memory space, however, NSC Tiny BASIC does not require that spaces separate (delimit) parts of the statement. It looks for other clues which are specified to the command keyword. These delimiters are discussed as each command is defined in detail later.

Occasionally, it will be necessary to use spaces after a HEXADECIMAL number to avoid confusion about where the number ends and the next keyword begins.

EXAMPLE

```
50IFA<#14FORI=1TO10

will be interpreted as having the number #14F
and, therefore, should be written:

50IFA<#14 FORI=1TO10
        or
50IFA<#14THENFORI=1TO10
```

The argument portion of a statement may be an expression or, in some cases, another statement. An expression specifies a number of a computation resulting in a number. Elements off expressions are discussed later.

2.1.4 The NSC Tiny BASIC EDITOR

NSC Tiny BASIC supports interactive debugging with a self-contained line editor. It also allows elimination of typing and other errors as the program is entered. Editing is done in the immediate mode. To print a program currently contained in memory, give the command LIST. Then examine the program and make changes and additions using the techniques described below.

NSC Tiny BASIC stores the program lines in line number sequence. If a line is typed with the same number as a line already in memory, the new line replaces the old one. If only the line number is entered, the line is deleted from memory, the only way to change it is to retype the line.

Until the carriage-return key is pressed at the end of the line, the characters entered are temporarily stored in a line buffer. If an error is detected in a line before it is stored in memory, correct it by backspacing through the line buffer to the mistake and retyping. Backspacing by pressing the backspace key or by holding down the control key and pressing H. Each backspace keystroke deletes one character from the line buffer. If more backspaces are entered than there are characters in the line buffer, NSC Tiny BASIC deletes the whole line

If it is necessary to delete a whole line before entering it in memory, it is quicker to hold down the control key and press the U than to backspace through the line buffer. The control U keystroke cancels the contents of the line buffer.

Although the editor is most useful for changing program lines, it can correct an immediate command before it is executed. It can also correct any user input required during a program run.

2.2 ELEMENTS OF A NSC TINY BASIC EXPRESSION

2.2.1 Introduction

Expressions represent the numeric values NSC Tiny BASIC needs to perform a task. An expression consists of one or more of the following elements:

- o constants
- o variables
- o operators
- o memory references
- o functions

The elements in a single expression are evaluated together when the statement is executed. The evaluation produces a single numeric value to be used in the execution to the instruction.

2.2.2 Constants (also Referred To As DATA)

A constant is a value that does not change during the execution of a program and must be represented by a number or string. In NSC Tiny BASIC all data is of two major types:

Numerics:	INTEGER	-32767 to +32767
	BYTE	0-255
	BOOLEAN	TRUE or FALSE
Strings:	STRING	Any alphanumeric character or "string" of characters

There are three numeric types of data: integer, byte and Boolean. All three of the numeric types can be intermixed in expressions without generating an error message. This can lead to unexpected results in complex expressions. NSC Tiny BASIC does generate an error message if string types are used where numeric types are expected but no error is generated if numeric types are used where string types are expected because numbers can be strings too.

2.2.2.1 Numbers -

All of our normal day to day usage of numbers are what mathematicians call the decimal number system. Any number we use is made up of digits. These digits are the numerals 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. All numbers we use are combinations of these ten basic digits.

Computers, due to the nature of the electronic circuitry that make them up, work best with a number system that has only two numerals, 0 and 1. This is called the binary number system. The NSC Tiny BASIC, however, allows us to use the decimal numbers and interprets these decimal numbers onto the binary numbers which the computer can use.

For those of us who are writing programs that must control or utilize electronic circuits directly from the computer, it is sometimes very important to know what binary numbers the computer is using for the task at hand (see chapter three for a discussion on binary numbers). However, binary numbers can get very long, for instance 987 would look like 1111011011. And, to make matters worse, it is not very obvious, from looking at the decimal number, what its binary representation is. To help programmers visualize the binary numbers the computer is using without having to write out all those 1's and 0's Tiny BASIC will also recognize what is called "hexadecimal" numbers. Hexadecimal numbers are composed of digits taken from sixteen numerals:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.

The numerals used to represent the numbers ten through fifteen are A thru F. This may cause some confusion at first but believe it or not you can get used to reading and writing these types of numbers.

If you wish Tiny BASIC to interpret a particular number as a "hexadecimal" number then that number must be preceded by a "#" symbol.

Example:

```
LET A=   987   Normal or decimal number
LET B= #1733   Interpret 1733 as a "hex" number
LET C=   3A3   Bad Number, You will get a syntax error.
```

Sometimes, in this manual, we will use hexadecimal numbers when the HEX number is easier to remember and use, for instance, the command NEW #1100 from chapter 1 could have been stated with a decimal number just as well; New 4352. The hexadecimal #1100 is easier to remember and use than 4352 because the hexadecimal numbers of most interest in addressing major blocks of memory, such as #1100, look more like boundary numbers than the decimal number 4352. This manual will go into more detail on the HEX number system in the next chapter though examples showing HEX numbers will occasionally appear in this chapter.

NOTE

It is not within the scope of this manual to go into great detail with the definition of certain terms, so forgive us if the definitions given seem somewhat brief.

2.2.2.2 Integers -

Integers are whole numbers such as 1, 5, -17, 0, etc. They do not have fractional or decimal parts. NSC Tiny BASIC considers integers to be whole numbers in the range of -32768 to +32768 (#0 to #FFFF in HEX).

Example:

3426	OK
-1872	OK
7 1/2	Not an integer, if used will cause error message.
32000	OK
-18.65	Not an integer. Error message generated.
-176000	Number too big. Error message generated.
#4F00	OK
#3D0F2	Number too big. Error message generated.

Normally numbers can be of any size but Tiny BASIC must restrict the sizes to the numbers if can handle to keep the NSC Tiny BASIC Program memory requirements within reason. If you have a need to use large numbers or fractions the NSC Tiny BASIC instructions can be used to handle them by writing the routines do so in your program.

2.2.2.3 Bytes -

Bytes are whole numbers in the range of 0 to 255 (#0 to #FF in HEX). Tiny BASIC further restricts numbers in a few places to bytes which we will mention as we progress through this chapter. (See Chapter 3, Product Description for more detail.)

Example:

0	OK
17	OK
#82	OK
-30	Minus not recommended (interpreted as 225)
1742	Too big (interpreted as 206)

2.2.2.4 Booleans -

Boolean numbers are a very special case of mathematics. They have only two values, usually called TRUE or FALSE. In Tiny BASIC FALSE is represented by a numeric value of 0 while TRUE is any other integer value. This may seem confusing but boolean numbers play a very important role in NSC Tiny BASIC, so just file this definition away until later in this chapter.

2.2.2.5 Strings -

String in the computer world has a very specific meaning. Consider, in English, the use of string to mean a chain of similar objects, such as a string of beads or a string of cars, etc. In Tiny BASIC a string is a "chain" of characters where a character is a letter, a number or special symbols.

Example:

```
"HI THERE"
"THIS IS A STRING"
"SO IS * - @ , + / ? ALL OF THIS"
"ARE YOU GETTING THE IDEA?"
```

NOTE

The string is the characters between the quotation (") marks but not including the quotation marks.

Another characteristic of strings is that they must have a beginning and an end. NSC Tiny BASIC interprets a quotation mark as the beginning of a string and the next quotation mark ends the string. There are other ways to begin and end strings but more on that later. The string enclosed by quotes is called a STRING CONSTANT.

2.2.3 Variables

As the name implies, a variable is a number (or string) that does not have a fixed or constant value. Since it does not have a fixed value then we must assign our variable a name. This name is called the variable identifier. In NSC Tiny BASIC there are twenty six variables available. They are simply given the "names" of the letters of the alphabet (i.e. "A" through "Z"). Numeric variables are normally used to hold the results of mathematical operations.

Example:

```
A= 15           Assign 15 to variable A
B= A+42         Assign 57 to variable B [15+42]
C= (A+12)*B     Assign 1539 to variable C [(15+12)*57]
A3= 32          Error, only variables A through Z are available
```

String variables are also available in NSC Tiny BASIC. They are identified by preceding the variable name with a "$".

NOTE

CAUTION! If you are already familiar with the BASIC language from another computer -BE CAREFUL- because in other versions of BASIC the "$" may follow the variable name.

Example:

```
10 $A= "HI THERE"    Assign HI THERE to string variable A.
20 $B= $A            Assign HI THERE to string variable B.
30 A$= "CAREFUL"     Error the $ must precede the variable name.
```

There is more to the string variables which will be covered in a more appropriate section.

Before any strings are stored the address of the string variables must be set to some free memory area. Care must then be taken to insure that a string loaded to a string variable is not too long thus causing the string to write over data stored in the addresses above those assigned to the original variable.

String moves are performed by a simple assignment statement such as:

Example:

```
$D = "HI THERE":$C=$D
```

The string variables $C and $D each now hold the same strings. The do not just point to the same locations in memory.

Example:

```
10 A=TOP             Points to ram memory above top of program
20 C=TOP+100         C points to RAM 100 bytes above A
30 D=TOP+200         D points to RAM 100 bytes above C
40 INPUT $A          Stores characters where A points
50 PRINT $A          Prints characters pointed to by A
60 LET $C="IS THE STRING INPUT AT LINE 10"
70 $D=$C             Stores characters where D points
80 PR$D              Prints characters pointed to by D
```

Before we move on there is one more type of variable to mention. This is the MEMORY VARIABLE (or MEMORY REFERENCE). This type of variable will be treated extensively in later chapters, but for now, this is a very powerful feature of NSC Tiny BASIC which allows the program to directly manipulate specific memory locations and INPUT/OUTPUT circuits. A memory reference is indicated to Tiny BASIC by preceding an expression with an ’at sign’ "@".

Example:

```
        10 A= @4500          Assigns contents of memory location 4500 to A
        20 B= 4501           Assigns 4501 to B
        30 @B= 17            Places 17 in memory location 4501
        40 C= @#8000         Contents of memory location#8000 to variable C
```

Memory references are of data type ’byte’, more on this later.

2.2.4 Operators

An operator indicates a calculation to be performed when an expression is evaluated. NSC Tiny BASIC supports three sets of operators: arithmetic operators, logical operators, and relational operators.

3.3.4.2 Arithmetic Operators -

NSC Tiny BASIC recognizes the traditional four (4) arithmetic operators which are:

+ Add
- Subtract
* Multiply
/ Divide

Operations are performed from left to right. If all four appear in a single expression, multiplication and division are performed first, followed by addition and subtraction. This may be altered by the use of parentheses.

Example:

```
3*24-18/3+10 = 76
3*(24-18)/(3+10) = 1
```

NSC Tiny BASIC does not support fractional numbers, therefore, the remainder of the division in the second line is discarded. Division by zero causes an error break and return to the immediate mode.

2.2.4.2 Logical Operators -

In addition to the traditional arithmetic operators, NSC Tiny BASIC also supports three (3) logical operators:

AND

OR

NOT

The AND and OR operators are very powerful operators for manipulating data that is to be used to control input and output peripherals. Their operation cannot be simply stated with examples at this point in the manual since we have not gone into the definition of bits yet, so we will leave the further definition of these operators to chapter 3.

NOT is an operator which performs a ONES COMPLIMENT operation on a single expression. What this means in plain english is that any expression that is preceded by a NOT will have one added to its result and then its arithmetic sign will be changed:

EXAMPLE:

```
NOT 15 yields  -16
NOT(-1) yields  0
```

2.2.4.3 Relational operators -

In NSC Tiny BASIC relational operators normally specify conditional relationships in IF statements and in DO statements. There are six (6) which are:

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

The symbol >< is not legal.

The above operators always return a boolean result (TRUE) or (FALSE).

Their usage is usually quite clear by their context (see examples of the IF and DO statements later in this chapter).

2.2.4.4 Complex expressions -

The order of analysis of expressions in NSC Tiny BASIC is:

1. Expressions are operated on from LEFT to RIGHT
2. Solution of expressions held in parenthesis
3. Solution of NOT, multiplication and division
4. Solution of AND, OR, addition and subtraction

EXAMPLE:

```
X = X + A OR C
A = X/Y
A = A + B OR C / Y
```

In the first example, above, the expression will be evaluated from left to right since from the list of priorities addition and OR are on the same line. In the second example the value of A will be determined by the value in X and the value in Y. In the third example A will be added to B, then C will be divided by Y, then the result of A+B will be ORed with the result of C/Y. The third example result will be significantly different than the result of the second example expression.

If there is ever any doubt about how your expression is going to be evaluated use parenthesis liberally in the expression.

Take care, in your expressions, to watch for conditions where division by zero might occur or where the results of an expression may become too large thus causing an erroneous result.

EXAMPLE:

```
FOR I = -4 to 4:  PR 10/I:  NEXT I
Will BREAK when I = 0

FOR B = 25 TO 38 :  A = B * 1000 :  PRINT A
Will print incorrect answers when B > 32
```

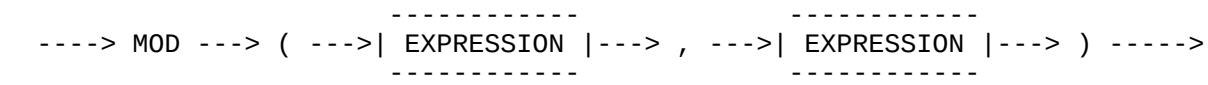
2.2.5 Functions

NSC Tiny BASIC includes the following functions:

1. MOD
2. RND
3. STAT
4. TOP
5. INC
6. DEC

2.2.4.1 The MODulo function -

SYNTAX



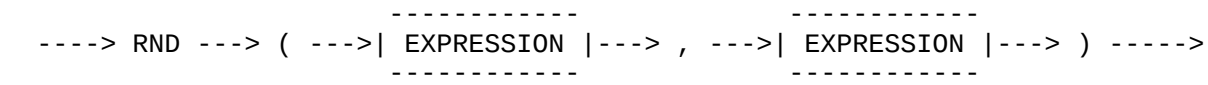
Returns the absolute value of the remainder of the first entry divided by the second entry where the entries are arbitrary expressions. If the value of the second entry is zero, an error break will occur as in any division by zero.

Example:

```
10 PRINT MOD(17,15)
yields 2
```

2.2.5.2 The RaNDom function -

SYNTAX



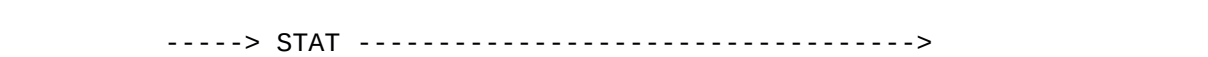
Returns a pseudo-random integer in the range of the first entry through the second entry, inclusive. For the function to perform correctly the first entry should be less than the second entry and the second entry minus the first entry must be less than or equal to 32767 (base 10).

Example:

```
10 PRINT RND(1,5)
yields either 1,2,3,4 or 5
```

2.2.5.3 The STATus function -

SYNTAX



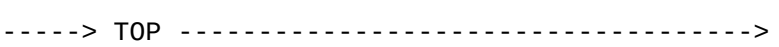
Returns the 8-bit value of the INS 8073 Status Register. STAT may appear on both sides of an Assignment Statement. This allows the programmer to modify the status register as well as read it. (More on this in chapter 3).

Example:

```
10 PRINT STAT
yields an integer between 0 and 255
```

2.2.5.4 The TOP OF PAGE FUNCTION -

SYNTAX



Returns the address of the first byte above the NSC Tiny BASIC program in the current page which is available to the user. This will be the address of the highest byte in the NSC Tiny BASIC program plus 1. All the memory in RAM above and including TOP can be used by the NSC Tiny BASIC program as scratchpad storage.

Example:

```
10 D=TOP
Stores the top of memory address in D
```

2.2.5.5 The INCrement function -

SYNTAX

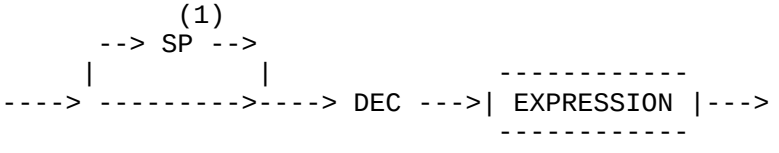
Used to increment a memory location (X) indicated by the expression.

Example:

```
INC #16FF
```

2.2.5.6 The DECrement function -

SYNTAX



Used to decrement a memory location (X) indicated by the expression.

Example:

```
DEC #16FF
```

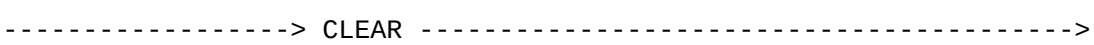
2.3 STATEMENTS

NSC Tiny BASIC includes the following statements:

1. CLEAR
2. DELAY
3. DO
4. FOR
5. GOSUB
6. GOTO or GO
7. IF / THEN
8. INPUT
9. LET
10. LINK
11. NEXT
12. ON
13. PRINT or PR
14. REM
15. RETURN
16. STOP
17. UNTIL

2.3.1 CLEAR

SYNTAX



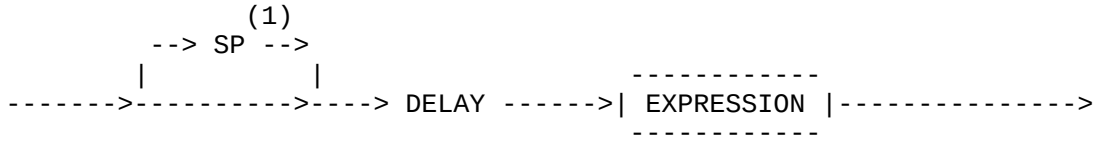
Example:

```
20 CLEAR
```

CLEAR initializes all variables (A to Z) to 0, disables interrupts, enables BREAK from the console and resets all stacks used by GOSUB, FOR/NEXT and DO/UNTIL statements and re-establishes the BAUD RATE from the jumpers on the board (see chapter 1).

2.3.2 DELAY

SYNTAX



Example:

```
20 DELAY 0
30 DELAY 17
```

DELAY is used to suspend the execution of NSC Tiny BASIC programs for a specific amount of time from 1 to 1040 milliseconds in millisecond increments. A time unit of 0 gives the maximum delay of 1040 milliseconds.

2.3.3. DO

SYNTAX

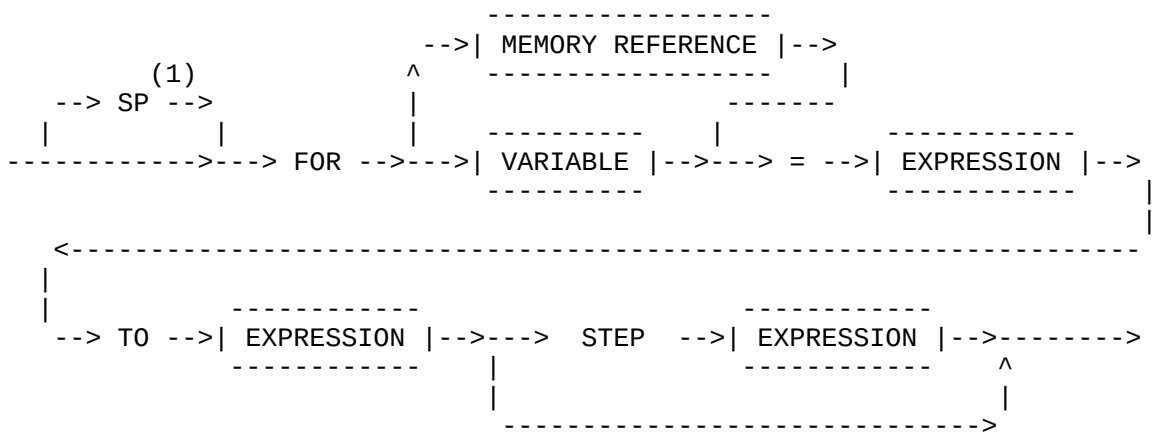
Example:

```
10 I = 0
15 DO
20 PRINT I
30 I = I + 1
40 UNTIL I > 10
```

DO is not a standard BASIC statement. It may be used in NSC Tiny BASIC instead of FOR / NEXT statements for program loops and greatly improves the readability of the program. The DO statement marks the beginning of a series of statements which are terminated with an UNTIL statement. The series of statements are repeated until a condition is met (see the UNTIL statement).

2.3.4 FOR / TO / STEP

SYNTAX



Example:

```
        FOR I=1 TO 100
        FOR J=A TO 55
        10 FOR Q=Z TO Z+100
        34 FOR K=1 TO 100 STEP 2
```

This is a standard BASIC statement where the STEP instruction is optional and if not included takes the value of +1. STEP may be either positive or negative. The FOR statement marks the beginning of a series of statements which are terminated with a NEXT statement (see the NEXT statement). The FOR statement must include the upper and lower limits for the loop. The loop is repeated until the upper limit of the FOR instruction is reached at which time the program will proceed to the numbered program statement following the NEXT statement. NSC Tiny BASIC causes a break if the variable in the NEXT instruction does not match that in the FOR instruction. FOR/NEXT statements may be nested up to four deep but each nesting level must be complete within a higher nesting level. A FOR loop will always be executed at least once.

2.3.5 GOSUB

SYNTAX

Example:

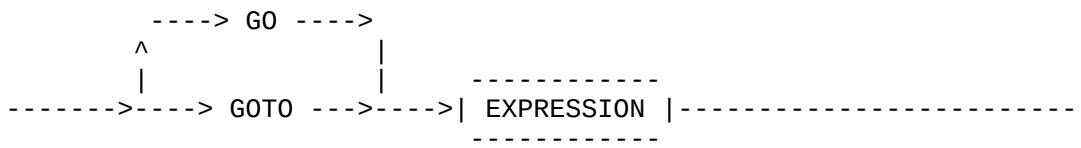
```
GOSUB 100
GOSUB A
10 GOSUB A*100
```

GOSUB is useful when a computation or an operation must be performed at more than one place in a program. Without having to enter the same code more than once the GOSUB statement may be used to call the often used routine at any time. The GOSUB statement is also useful since it allows programs to be written in small pieces (called "procedures" or "subroutines") which makes for easier finding and correcting of errors. The expression following the keyword GOSUB may be either the number of the first line of the subroutine or an expression which results in the line number of the desired subroutine.

One subroutine may call another. The last instruction of each "subroutine" must be the RETURN instruction which causes the NSC Tiny BASIC program to resume execution at the line number following the original GOSUB instruction. In this way, subroutines may be nested to a level of eight (8).

2.3.6 GOTO / GO

SYNTAX



Example:

```
GOTO 100
45 GO   100
GOTO #FF
56 GOTO A*100
```

GOTO unconditionally changes the sequence of the program execution. NSC Tiny BASIC allows the program to branch to a specific line number or a line number called by an arbitrary expression. Therefore, in the example GOTO A*100, if A is 10, 11 or 12 the program will continue execution at program steps numbered 1000, 1100, or 1200.

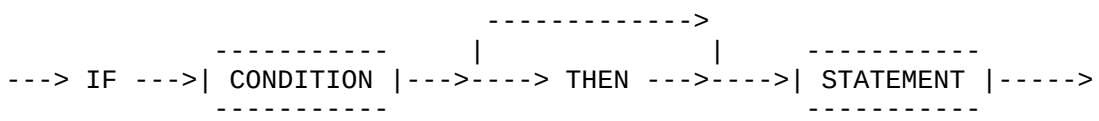
GOTO is often used in interactive debugging because GOTO enters the RUN mode. Unlike the RUN command GOTO can specify the line number to begin execution and does not reset the variables to zero (0).

Because GOTO unconditionally changes the program sequence, any statements that follow on the same line will not be executed.

"GO" may be used to make more compact programs, although this may make them less readable.

2.3.7 IF / THEN

SYNTAX



Example:

```
IF A>B THEN PRINT "A>B"
IF A>B PR "A>B"
IF X=Y IF Y=X PRINT "X=Z"
IF A<>B I=0:J=K+2:GOTO 100
IF A<1 GO 1000
```

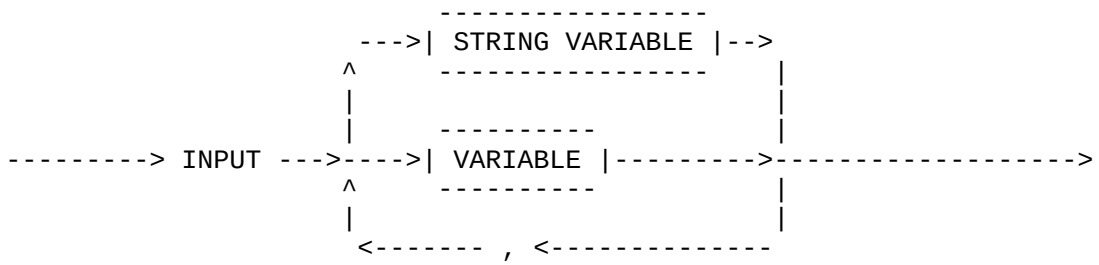
The IF/THEN statement allows the program flow to be modified by a logical test condition. The test condition follows the IF instruction of the statement and may be either a line number, or a list of statements separated by colons.

IF compares the value of the first expression to the value of the second expression. If the result of the comparison is TRUE (-1) the clause of the IF/THEN statement is executed. If the comparison is FALSE the next line numbered statement is executed.

NSC Tiny BASIC allows the omission of the word THEN to conserve memory space.

2.3.8 INPUT

SYNTAX



Example:

```
5  INPUT A
15 INPUT A, B
25 INPUT $C
```

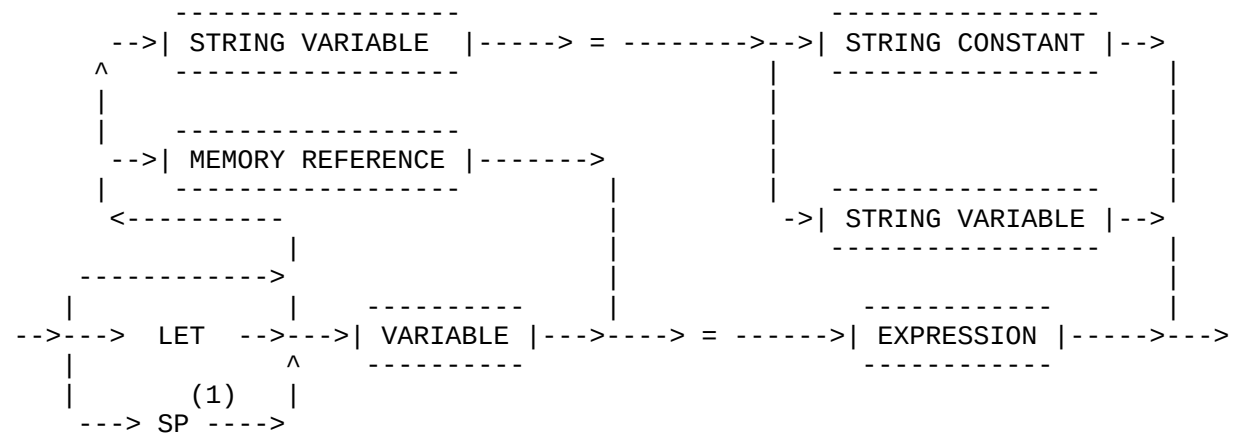
INPUT is used to input data to an NSC Tiny BASIC program. One or more items (variables, expressions and strings), separated by commas, may be entered. Upon execution of the INPUT statement the NSC Tiny BASIC program will prompt the user with a question mark "?". Each INPUT statement must have one or more variable into which to put the items requested by the prompt "?". After the user responds with a carriage return the NSC Tiny BASIC program will check for proper data type. If no data type is encountered an error break occurs followed by the prompt ">".

The acceptable input for the variable "A" or "B" is an integer. If the input is not an integer the error message "RETYPE" will be printed followed by the prompt "?" until an integer or expression is read. A comma between multiple expression inputs may be replaced by spaces if the following expression does not start with a plus (+) or minus (-) sign.

The acceptable input for "$C" is any ASCII character (see ASCII character chart in the appendix) until a carriage return is typed or the number of characters limit for one line is reached. The entered characters are stored in a line buffer until a carriage return is pressed. The characters are then transferred to memory in successive locations beginning at "C" and continuing with "C+1" etc, until "C+n" which holds the end of string character, the carriage return "<cr>". Quotation marks are not used for input.

2.3.9 LET

SYNTAX



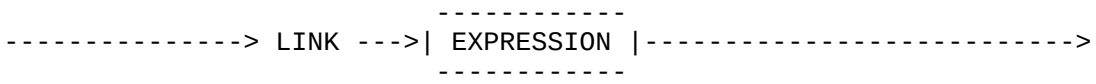
Example:

```
10 LET X=34
10 X=A*34
```

LET may be used or omitted in an assignment statement. The execution of an assignment statement is slightly faster if the word LET is used, however, it requires more memory. The left portion of an assignment statement may be a simple variable, number or an expression in parentheses.

2.3.10 LINK

SYNTAX

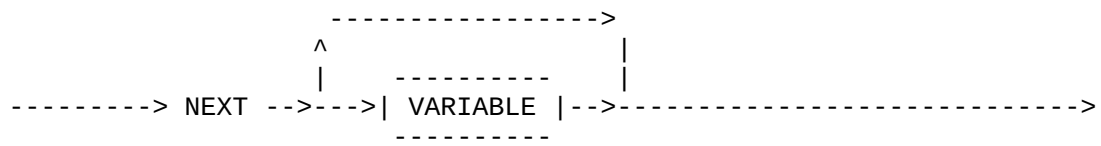


Example: 55 LINK#8D00
15 IF A=5 THEN LINK(B+15)
450 DO: LINK4500 : UNTIL A=32

The LINK statement is the means by which a program may call machine language routines. This is used when efficient code or time critical processes must be executed. Execution of this statement will cause the INS 8073 to transfer control to the machine language routine beginning at <address>. This statement is conceptually very similar to the GOSUB statement. The Tiny BASIC program will continue with the next statement upon a return from the LINK statement (See chapter 3).

2.3.11 NEXT

SYNTAX



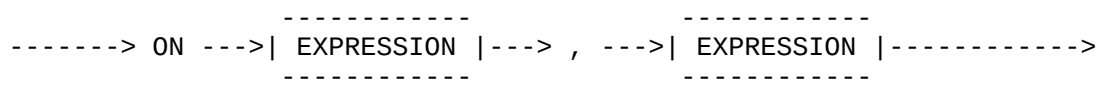
Examples:

```
95 NEXTJ
15 FOR I=0 TO 15 : PRINT "" : NEXTI
737 NEXT N : NEXT T : PR "LOOPS DONE"
```

The NEXT statement is used to mark the end of a FOR loop. The variable identifier may be omitted. If it is included, it must be the same identifier as specified in the FOR statement which this NEXT statement is terminating. It is recommended that the identifier always be used, it makes the program easier to read and the error checking done by Tiny BASIC is more likely to catch a programming error. (See the FOR statement).

2.3.12 ON

SYNTAX



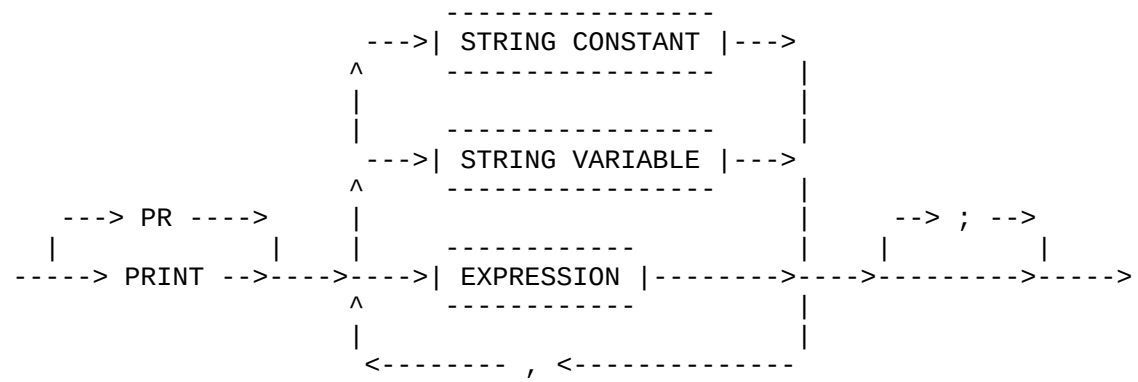
Example:

```
15 ON 1,0
30 ON 2,500
440 G=1500 : ON 1,G : B=#8500
```

The ON statement allows the NSC Tiny BASIC program to GOSUB to <line number> when a hardware interrupt occurs (See chapter 3). Specifying a line number of zero causes a software disable of the corresponding <interrupt number>. Use of the ON statement disables terminal interrupts (break function).

2.3.13 PRINT / PR

SYNTAX



Example:

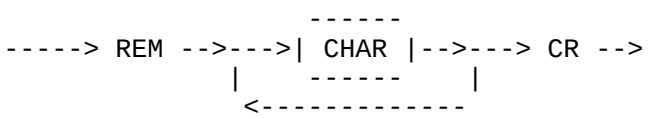
```
        10 $D = "HI THERE":PR$D
        yields HI THERE<cr><lf>
```

The PRINT statement is used to output information from the program. To conserve memory space a "PR" may be used in place of the word "PRINT". Quoted strings are displayed exactly as they appear with the quotes removed. If a message must be punctuated with a quotation mark, use the single quote or apostrophe instead. All numbers and results of expressions are printed in decimal format with leading zeros and divisional remainders omitted. Positive numbers will be preceded by a space and negative numbers will be preceded by a minus sign (-). There is a trailing space for all numbers. A semicolon (;) at the end of a PRINT statement suppresses the carriage return and line feed with which NSC Tiny BASIC normally terminates the output.

Strings stored in memory (such as those generated by string input and string assignment statements) may also be printed.

2.3.14 REMark

SYNTAX



Example:

```
10 REM THIS IS A COMMENT TO THE PROGRAMMER
95 X=35 : REM THE VARIABLE X IS SET TO 35
```

The REM statement is used to insert comments or explanatory messages in the code. NSC Tiny BASIC ignores anything following the REM keyword and, therefore, must be the last command on the line. Remarks make the program easier to read and can provide good documentation within the program listing but take considerable memory space and should be left out of the finished program to make best use of available memory.

2.3.15 RETURN

SYNTAX

Example:

```
50 RETURN
200 FOR I=0 TO 10 : PR "" : NEXTI : RETURN
```

The RETURN statement is always the last instruction of a subroutine. It does not require an argument, because the GOSUB stores the next statement where the RETURN can resume normal sequential execution. RETURN must be the last statement on a line. (Refer to the GOSUB statement).

If one subroutine calls another, the RETURN statement at the end of the second subroutine returns execution to the first subroutine. In this way, subroutines may be nested to the depth allowed by the memory available to the GOSUB stack (see Programming notes at the end of this chapter).

2.3.16 STOP

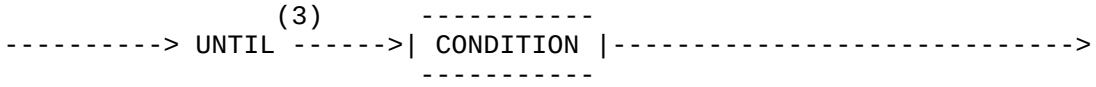
SYNTAX


```
Example:         25 STOP
         550 PRINT "WE ARE DONE" : STOP
```

The STOP statement may be used in a program to halt execution at any point which is useful for debugging purposes. When NSC Tiny BASIC encounters a STOP statement it prints a stop message and the current line number and executes a break. Execution of the program may be continued from the next statement following the STOP by typing the command CONT. (See interactive debugging and errors in this chapter).

2.3.17 UNTIL

SYNTAX



Example:

```
35 UNTIL X=15
90 DO: PRINT "": UNTIL (X+Y)=99
490 UNTIL A=0 : PR "THE TASK IS COMPLETE"
```

The UNTIL statement is the last statement in a DO loop. The DO loop will continue to execute until the expression in the UNTIL statement is true. Normal sequential execution by the program will then continue with the next statement after the UNTIL statement. (Refer to the DO statement).

2.3.18 Multiple Statement Line

Example:

```
10 FOR I = 1 TO 10:PR RND (1,10):NEXT I
```

Multiple statements are permitted on a single line by placing a colon ":" between statements. Lines may be made as long as the line buffer will permit (72 characters). This may help readability and save memory but care must be taken when writing each line, especially those containing conditional tests (see IF statement).

2.4 COMMANDS

As mentioned above NSC Tiny BASIC has two modes of operation, command mode and execute mode. Upon reset and any time the program execution is halted the Tiny BASIC is in command mode which it indicates by printing a greater than symbol ">". In the command mode most statements may be executed immediately (see errors and interactive debugging below). In addition there are four commands which are not executable in the RUN mode:

NEW

RUN

CONT

LIST

2.4.1 NEW

Example:

```
NEW #1100
NEW
NEW 32768
```

The NEW command has two forms, with and without an argument. If the NEW command is followed by an argument, the argument becomes the starting address in memory of a NSC Tiny BASIC program. If the argument points to a ROM address the NSC Tiny BASIC program beginning at this address will execute. If programs are going to be written into RAM then this command must be given before writing your program else an out of memory error will result. Program memory including the end of memory pointer is not altered by this command.

Once the starting address has been specified as above, then a NEW command is typed without an argument. This clears any old programs that may have been present in RAM and is now ready for you to enter a new program.

2.4.2 RUN

Example:

```
RUN
```

The RUN command starts program execution at the beginning of the Tiny BASIC program. It clears all variables and stacks, enables the break key on the terminal and resets the terminal baud rate per the jumpers.

2.4.3 CONTinue

Example:

```
CONT
```

The CONT command is used to resume execution of a program which has been halted with a STOP statement or the break key on the terminal. Execution will resume with the next executable statement after the STOP statement or the next executable line after the break. (See errors and interactive debugging below).

2.4.4 LIST

Example:

```
LIST
LIST 150
```

The LIST command is used to generate a listing of the current program stored in memory on the terminal. If an argument is specified the listing will begin at that line number.

2.5 ERRORS AND INTERACTIVE DEBUGGING

2.5.1 Errors

Errors occur whenever NSC Tiny BASIC is unable to interpret an instruction. An error returns the system to the immediate mode, except when an INPUT statement is expecting an integer, with all variables and stacks as they were at the time the error was detected. The error message will be displayed on the terminal in the following format:

ERROR <error code> AT <line number>

The error codes are shown below.

NSC Tiny BASIC ERROR MESSAGES

1. Out of memory
2. Statement used improperly
3. Unexpected character (after illegal statement)
4. Syntax error
5. Value (format) error
6. Ending quote missing from string
7. GO target line doesn’t exist
8. RETURN without previous GOSUB
9. Expression or FOR-NEXT or DO-UNTIL nested too deeply
10. NEXT without previous matching FOR
11. UNTIL without previous DO
12. Division by zero

If the error is found while the program is running, the error message will display the line number. An error message which occurs while executing an instruction in the immediate mode will not display a line number. An error occurs after a line is entered and the carriage return has been pressed if NSC Tiny BASIC cannot interpret a keyword or expression. NSC Tiny BASIC will stop executing on the first error found so lines with multiple errors must be debugged completely before the next line will be interpreted.

2.5.2 Interactive debugging

NSC Tiny BASIC allows interruptions and changes during a program run to correct errors and add new instructions without disturbing the sequence of the program.

To interrupt the running program, hold the <control> key down and press <C>. The input buffer is tested for the control C character at the beginning of each BASIC instruction. If <control C> is found program execution stops and the following message is displayed on the terminal:

STOP AT LINE <line number>

A more reliable method is to press the <break> key. This causes the program execution to stop at the beginning of the next numbered line. The message displayed on the terminal is:

BREAK AT LINE <line number>

Another method is to insert STOP statements where you wish to halt the program. The variables and program execution level is preserved, the program or variables may be displayed, then execution may be resumed by entering a CONT statement. There is a message which is displayed on the terminal as follows:

STOP AT LINE <line number>

As a last resort the program may also be stopped by pressing the reset switch, connected as described in the preceding chapter. The program pointer must then be reset with the NEW <program address> statement. The program stored in memory will not be affected.

2.6 PROGRAMMING NOTES

2.6.1 Programs in PROM

Due to the way Tiny BASIC works the first statement in a program to execute in PROM must be CLEAR. If CLEAR is not present the program will not function properly or not work at all.

2.6.2 Execution speed vs. memory space

2.6.2.1 Introduction -

Depending on the application, some programs are limited by the memory space available, while others are limited by the time necessary to execute each instruction. There are trade-off relationships between program readability, memory space, and execution speed. The following sections advise how to code a program for either minimum memory usage or minimum execution time.

2.6.2.2 Conserving Memory Space -

To conserve memory space, eliminate all optional keywords such as LET and THEN from the code, and eliminate all spaces wherever possible. Abbreviate PRINT to PR and GOTO to GO. A STOP command is implied at the end of a program so it may be omitted. Remarks should also be eliminated.

There are three ways to conserve memory space by reworking line numbers. First, line numbers are stored in ASCII, one byte per digit. Bytes are saved by using low line numbers.

Second, space is allotted for 26 variables whether they are used or not. Variables not used by the program may be used to address subroutines or GOTO destinations by storing the line number in the variable. For example if the subroutine at line 4000 is used several times in a program, the following statements store the line number in the variable, the use the variable as a line number. Example

```
23 A=4000
24 GOSUB A
```

The variable A could be used elsewhere in the program where the value 4000 is needed as data; NSC Tiny BASIC makes no distinction between data and line number values.

The third method is that each line has an overhead in memory of ASCII bytes. One for the line number and one for the terminating character <cr>. This overhead may be reduced to one byte per statement by entering several statements on a single line. The overhead is then one byte for the colon separating the statements.

2.6.2.3 Improving execution time -

Although NSC Tiny BASIC does not execute instructions as quickly as machine code, some coding practices improve execution time. Including the keyword LET, for example, eliminates the interpreter's search through the keyword list to find the implied command.

Speed of execution may also be improved by eliminating spaces, remarks and THEN, and also by abbreviating PRINT and GOTO. It takes longer to convert two more ASCII digits to binary than to fetch a variable from memory. Use variables for any frequently needed large constant.

Using low line numbers for frequently used subroutines saves executions time as well as memory space. When normal sequential execution is interrupted by a GOTO, GOSUB, or RETURN, NSC Tiny BASIC scans the program from the beginning until it finds the desired line. Therefore, the closer the desired line is to the beginning of the program, the sooner the search succeeds.

2.6.3 The nesting stack

DO/UNTIL statements may be nested up to 8 levels deep (including interrupt levels).

GOSUB statements may be nested up to 8 levels deep (including interrupt levels).

FOR / NEXT statements may be nested up to 4 levels deep (including interrupt levels).

The arithmetic expression stack is 13 levels deep.

CHAPTER 3

PRODUCT DESCRIPTION

3.1 INTRODUCTION

The MC-1N is a "minimum" configuration microcomputer that can be used in "real time" process control applications. The MC-1N is designed to be used where low cost application of the power and flexibility of a microprocessor are required. The MC-1N is compact and easy to use and affords many opportunities for controlling a myriad of different kinds of hardware through its 24 I/O lines. The MC-1N is contained on a 3" by 4" double sided printed circuit board which has the following major components:

INS 8073 microcomputer with built in Tiny BASIC
8255A Programmable Peripheral Interface
MM57184N Programmable Real Time Clock/Calendar
HM6116LP 2K by 8 RAM
2732 4K by 8 ROM
LM358 Serial Interface Buffers
ICL7660 Negative Voltage Supply

A more detailed description of the functions of the first four of the above components is included in later sections. First, however, is a discussion of how and what a computer uses in its execution of the program that controls it.

3.1.1 Bits and Bytes

This section assumes that you will be using an INS 8073 with at least 256 memory locations; this is the minimum configuration to run NSC Tiny BASIC.

The following points will be detailed:

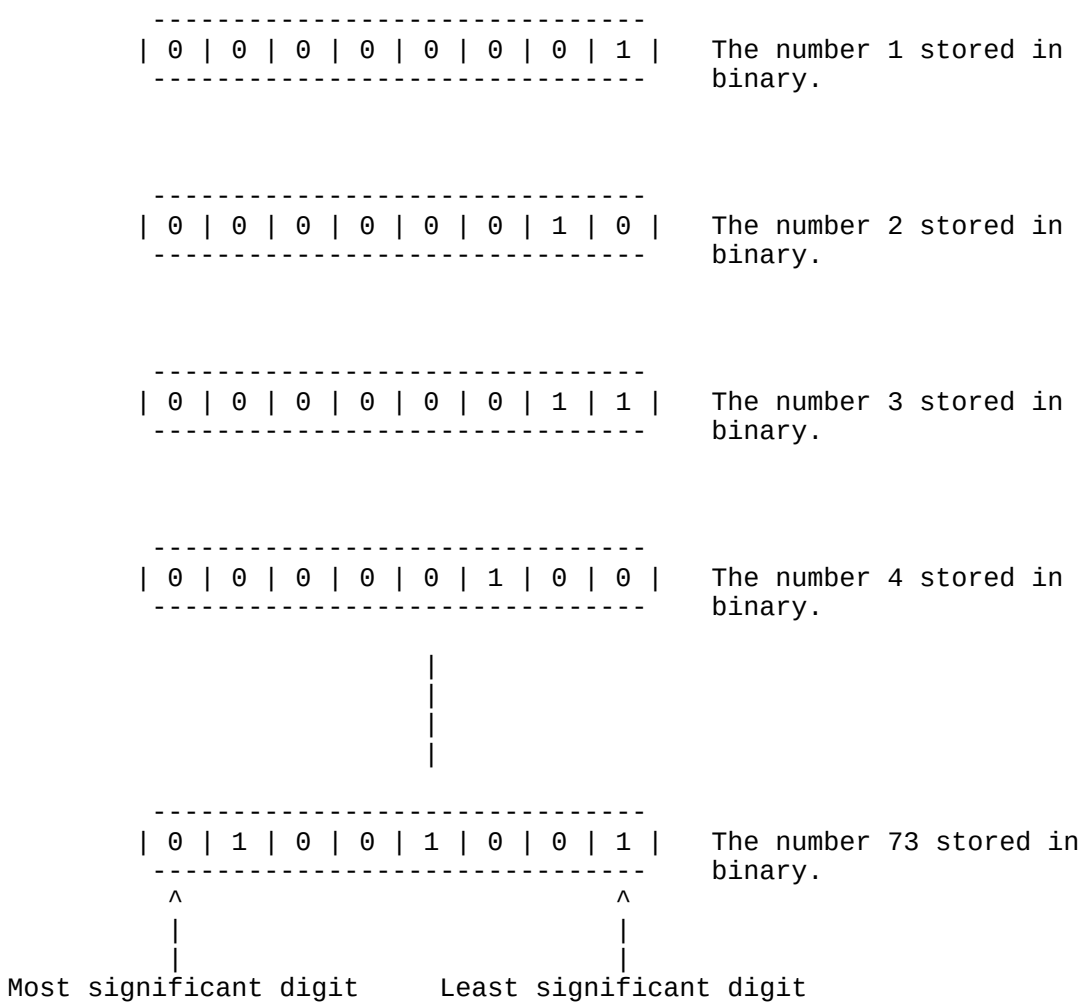
- A binary digit (bit) is either 0 or 1.
- One byte consists of eight binary digits commonly called bits.
- Each memory location holds, or stores, one byte of information.

Computers can deal with only two conditions 1 (ON) or 0 (OFF). (Humans have many levels of awareness such as in sensing temperature there are many sensational levels between cold and hot.) These two conditions are referred to,

individually, as binary digits or bits. Bit is a contraction of BInary digIT.

A computer with this level of sophistication is of little more use than an ordinary light switch. By combining these bits into groups a computer can then have more flexibility than on or off. A group of eight bits is referred to as a BYTE. Each memory location in the INS 8073 microcomputer is eight bits, or one byte, wide, and therefore holds one byte of information.

A memory location may be thought of as pictured below.



Each successively more significant bit has a higher value which allows a byte to hold the values of from 0 (all zeros) to 255 (all ones).

A computer that can count only to 255 is of rather limited use so bytes may be combined into larger groups as in the case of some special purpose memories called REGISTERS. One such register is called the program counter and consists of two bytes, which allows it to count to 65536. The program counter is used to select the memory address which holds the next piece of program information to be used by the computer.

3.1.2 The memory address

Each memory location has a unique numeric address. The NSC Tiny BASIC program in the INS 8073 system occupies locations with addresses 0 to 2559.

An expanded INS 8073 system might have more memory locations. For example, the MC-1N has 2048 locations but the INS 8073 will allow expansion up to 65536 locations. These may be memory locations or ports for peripheral devices.

Two points to keep in mind are:

- Memory locations 0 to 2559 hold NSC Tiny BASIC in the on-chip ROM (Read Only Memory) of the INS 8073.
- Addressed 2560 through 65471 are yours to use. When you type in an NSC Tiny BASIC program you use some of these. The larger your program the more you use. In using the Programmable Peripheral Interface you will also be using some of these addresses. Not all of these memory locations will actually be used for most applications and not all of them are connected in the MC-1N but they are available and may be used by decoding them in your application circuitry.

3.1.3 The hexadecimal number system

To make the task of dealing with the computers binary number system more manageable the hexadecimal number system is used. The hexadecimal (base sixteen) number system is a handy shorthand for talking about bits and bytes and memory addresses.

In hexadecimal, addresses range from #0000 to #FFFF.

The number sign (#) is used to tell you that the number is hexadecimal instead of decimal. This is the notation used in NSC Tiny BASIC; other notations exist in other literature.

The hexadecimal number system has more digits than the decimal system. To give symbols to these additional digits the letters A, B, C, D, E, and F are used to denote the hexadecimal equivalents of 10, 11, 12, 13, 14, and 15, respectively.

Just as in the decimal number system, each hexadecimal digit has a positional (or place) value. The digit occupying any position is multiplied by the value of that particular position. These products are then added together to obtain the value of the number.

Hexadecimal position values are expressed as powers of sixteen (rather than ten as in the decimal system). Positions are numbered from right to left according to the increasing powers:

POSITION 3	POSITION 2	POSITION 1	POSITION 0
3	2	1	0
16	16	16	16

The decimal values of the powers of 16 are:

```
   0          1             2              3
16  1      16  = 16      16  = 256      16  = 4096
```

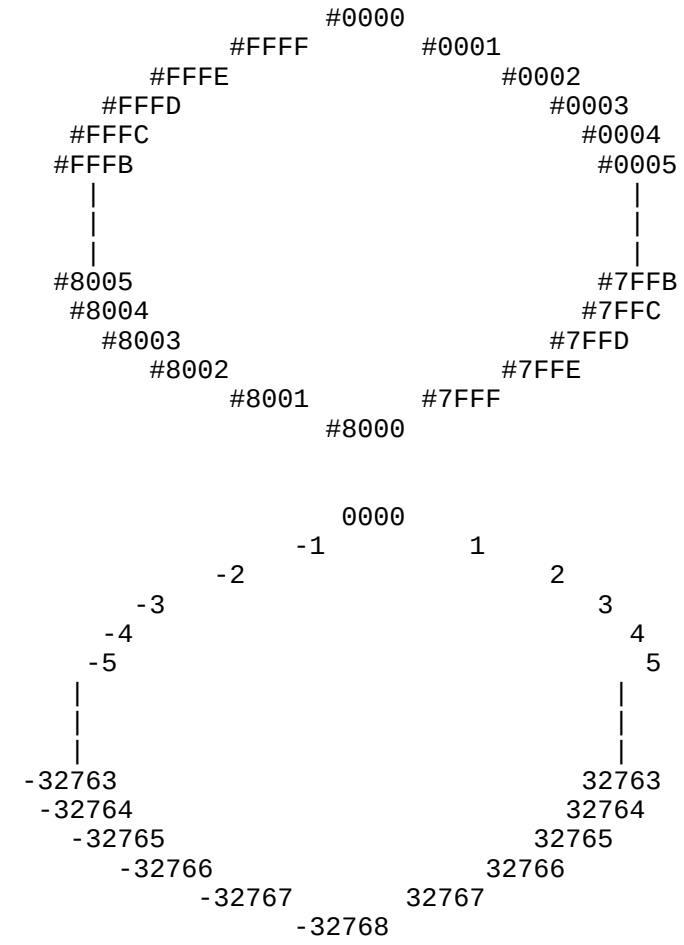
For example, the number #6A3F should be interpreted as:

```
                  3
       6      x 16 = 24576

                  2
      #A (10) x 16 =  2560

                  1
       3      x 16 =    48

                  0
     + #F (15) x 16 =    15
     ---------------------------
     #6A3F           = 27199
```

If we ask the INS 8073, in NSC Tiny BASIC, to print a hexadecimal number NSC Tiny BASIC will respond with a decimal equivalent.

Since NSC Tiny BASIC uses two bytes to store a number in any of its variables (A through Z) the largest number that can be stored is 65535. However, to be able to handle negative numbers the most significant bit is used as a sign bit. This means that numbers between #0000 and #7FFF are equivalent to decimal numbers 0 through 32767 and numbers #8000 through #FFFF are equivalent to decimal numbers -32768 through -1, respectively.

To more graphically illustrate what this means study the hexadecimal and decimal number circles below:



3.1.4 The AND, OR, NOT OPERATORS

Now that you have a handle on bits and bytes we can describe in more detail the logical operators NOT, AND, OR which are extremely useful for the software control of hardware.

All of these operations operate on 16 bit words.

The NOT operator performs a bit complement on the factor following the operator. A bit complement simply changes a bit from whatever state it is in to the opposite state.

EXAMPLE

```
        A = 0 0 1 0  0 0 1 0  0 0 1 0  0 0 1 0 = #2222 =  8738

    NOT A = 1 1 0 1  1 1 0 1  1 1 0 1  1 1 0 1 = #DDDD = -8739
```

The NOT operator is usually used with conditionals to invert the sense of the condition.

EXAMPLE

```
    20 IF A = 15 PR "A IS EQUAL TO 15"
    40 IF NOT(A = 15) PR "A IS NOT EQUAL TO 15"

    Z = 48
    55 D0
    65 Z = Z - 1
    75 UNTIL NOT(B = Z - 48)
```

The AND operator uses two terms just as do add and subtract.

EXAMPLE

```
10 A = Z AND 15
40 T = C AND D
45 IF (A AND B) = 0 THEN GOTO 15
```

The AND operator does a logical AND on each set of bits according to the following table:

```
1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0
```

Therefore:

```
          A = #0F0F = 0000 1111 0000 1111
          B = #1234 = 0001 0010 0011 0100
A AND B = #0204 = 0000 0010 0000 0100
```

The AND operator is most useful for getting rid of undesired bits in input and output routines or for setting desired bits to 0, or in complex conditional expressions:

EXAMPLE

```
15 A = A AND #FF :  REM will force A to be between 0 - 255.
43 T = C AND #FE :  REM will force bit 0 of C to 0.
97 IF (A = B) AND (C = D) THEN PR "MATCHED"
```

The OR operator uses two terms as does AND.

EXAMPLE

```
 15 A = B OR #F0
 22 T = (C OR D) + (T OR 1)
210 UNTIL (B = 0) OR (Z = T)
```

The OR operator does a logical OR on each set of bits according to the following table:

```
1 OR 1 = 1
1 OR 0 = 1
0 OR 1 = 1
0 OR 0 = 0
```

Therefore:

```
        A = #0F0F = 0000 1111 0000 1111
        B = #1234 = 0001 0010 0011 0100
A or B = #0204 = 0001 1111 0011 1111
```

The OR operator is most useful for setting desired bits to 1, or in complex conditional expressions:

EXAMPLE

```
15 A = A OR 8 :  REM sets only bit 3 of A high.
34 DO :  GOSUB 500 :  UNTIL (A = 0) OR (A = 12)
```

3.1.5 MC-1N MEMORY ORGANIZATION

The MC-1N has 2000 bytes of local memory (RAM). The first 256 bytes are reserved for Tiny BASIC variables and other house-keeping functions. The remaining 1792 bytes can be used for programs #1100 to #17FF.

MC-1N MEMORY MAP

#0000 - #09FF	Internal 8073 ROM (2.5K)
#0A00 - #0FFF	Not Available to MC-1N
#1000 - #10FF	Tiny BASIC Variables
#1100 - #17FF	MC-1N Program Space
#1800 - #7FFF	Not Available to MC-1N
#8000 - #8FFF	Users Program ROM
#9000 - #A7FF	Not Available to MC-1N
#A800 - #AFFF	Real Time Clock/Calendar
#B000 - #B7FF	Not Available to MC-1N
#B800 - #BFFF	Programmable Peripheral Interface
#C000 - #FCFF	Not Available to MC-1N
#FD00 - #FDFF	Baud Rate Select
#FE00 - #FFBF	Not Available to MC-1N
#FFC0 - #FFFF	Internal 8073 MPU RAM (64 bytes)

3.2 PARALLEL I/O LINES

The MC-1N has 24 programmable Input/Output lines available on J2. See section 3.5 page 66 for more information. These inputs and outputs are implemented using an 8255 programmable parallel interface chip. Depending on your applications the PPI can be used for a variety of tasks such as, printer drivers, display drivers, keyboard inputs, relay drivers, D/A converter inputs, A/D converter control, etc. The PPI is an extremely versatile microprocessor I/O device and is beyond the scope of this manual to go into all of its details. If you would like to explore the PPI capabilities beyond those specified in this section, refer to the INTEL Component Data Catalog or "Microcomputer interfacing with the 8255 PPI chip" by Paul F. Goldsbrough, published by Howard W. Sams Co.

The PPI occupies four sequential memory locations in the MC-1N:

#B800	Port A
#B801	Port B
#B802	Port C
#B803	PPI control register

For most applications the PPI will be used in what is referred to in the data sheets as mode 0. This being the case, we will limit our treatise on the PPI to the mode 0 operation

3.2.1 Configuring the PPI

The PPI has 24 I/O lines which may be specified as inputs or outputs. These 24 lines are organized as three groups of 8 lines each called port A, port B, and port C. The PPI control register is used to define what lines are to be inputs and what lines are to be outputs. Ports A and B may be specified as all inputs or all outputs, but, Port C may be specified as all inputs or all outputs or half inputs and half outputs. These options are set by setting #B803 to the following:

ADDRESS DATA	PORT A	PORT B	PORT C LOWER 4 BITS	PORT C UPPER 4 BITS
@#B803 = #80	outputs	outputs	outputs	outputs
@#B803 = #81	outputs	outputs	inputs	outputs
@#B803 = #82	outputs	inputs	outputs	outputs
@#B803 = #83	outputs	inputs	inputs	outputs
@#B803 = #88	outputs	outputs	outputs	inputs
@#B803 = #89	outputs	outputs	inputs	inputs
@#B803 = #8A	outputs	inputs	outputs	inputs
@#B803 = #8B	outputs	inputs	inputs	inputs
@#B803 = #90	inputs	outputs	outputs	outputs
@#B803 = #91	inputs	outputs	inputs	outputs
@#B803 = #92	inputs	inputs	outputs	outputs
@#B803 = #93	inputs	inputs	inputs	outputs
@#B803 = #98	inputs	outputs	outputs	inputs
@#B803 = #99	inputs	outputs	inputs	inputs
@#B803 = #9A	inputs	inputs	outputs	inputs
@#B803 = #9B	inputs	inputs	inputs	inputs

3.2.2 PPI Outputting

Once the PPI is configured for input and output it is a simple matter to read and write to the I/O lines. See section 3.5 page 66 for more information. If, for example, port A was configured to be an OUTPUT, you may test it by connecting a voltmeter to one of the port A pins on connector J2 and typing:

```
@#B800 = 0              : REM The voltmeter should read near 0 volts

@#B800 = 255            : REM The voltmeter should read near 5 volts
```

The above statements set all pins on the port A to low then high. If you only wanted to set bit 0 of port A low then high (pin 4 on J2) then the above example should be changed to:

```
@#B800 = 0

@#B800 = 1
```

Since the port is now configured and tests correctly, let's write a simple program to turn the port A bit 0 on for one second then off for one second continuously:

```
10 A = #B800
20 @A = 0 :  DELAY 1000
30 @A = 1 :  DELAY 1000
40 GOTO 20
```

This program will run forever or until a BREAK key or CONTROL C key is pressed on the terminal.

In actual applications each output bit of the PPI may be assigned different tasks. Also, when you change one of the bots you may not want the other bits to change. Since the PPI outputs cannot be read back, your program must keep a storage register with the current output pin states.

EXAMPLE:

```
10 M = 0
20 PR"PIN 4 = ";:INPUT A
30 IF A = 0 THEN M = M AND 254 :  GOTO 50
40 M = M OR 1
50 @#B800 = M
60 PR"PIN 3 = ";:INPUT A
70 IF A = 0 THEN M = M AND 253 :  GOTO 90
80 M = M OR 2
90 @#B800 = M
100 GOTO 20
```

In the above example M is the storage register of the PPI outputs.

3.2.1 PPI Inputting

Once a port has been configured for inputting, the input pins may be read with simple memory reference statements. For instance, if Port A is configured as an input port then PR @#B800 will print out the decimal value of the eight input lines on Port A.

Wire connector J-2 pins 4, 3, 2, 1, etc. to ground (pin 6), then PR @#B800 will print a 0. See section 3.5 page 66 for more information.

Wire connector J-2 pin 4 to +5vdc (pin 5) leaving the others on ground then PR @#B800 will print 1.

Wire a DPST switch with the switch common to connector J-2 pin 4 and the other two switch terminals to +5vdc and ground, respectively. The following program will print on the terminal the position of the switch.

EXAMPLE

```
10 A = # B800
20 IF @A = 0   PR"SWITCH IS OFF" : GOTO 40
30 PR"SWITCH IS ON "
40 GOTO 20
```

In actual usage many inputs, say eight switches, will be connected to the input port. If you are only interested in the position of the switch connected to connector J-2 pin 1 the above program could be written:

EXAMPLE

```
10 A = # B800
20 IF (@A AND 8) = 0   PR"SWITCH IS OFF" : GOTO 40
30 PR"SWITCH 1 IS ON "
40 GOTO 20
```

3.3 ADDITIONAL I/O AND INTERRUPTS

3.3.1 Inputs and Outputs using STAT FUNCTION

The MC-1N has five additional I/O lines available on connector J-1. See section 3.5 page 66 for more information. These five lines come directly from the INS 8073 processor chip and may be accessed using the STAT function in Tiny BASIC.

These are:

SA	SENSE A	INPUT	J1-1
SB	SENSE B	INPUT	J1-5
F1	FLAG 1	OUTPUT	J1-2
F2	FLAG 2	OUTPUT	J1-8
F3	FLAG 3	OUTPUT	J1-7

If a terminal is used, SA and F1 will be occupied with serial I/O. They may be used for other functions but the circuitry must disable the terminal I/O to avoid interference.

On the MC-1N, SB, F2 and F3 are available.

F2 and F3 can be set and cleared with:

```
STAT = STAT AND #0D
```

SB may be read with:

```
PR STAT AND #20
```

3.3.2 INTERRUPTS

The SA and SB inputs can also be interrupt inputs. Interrupts are used to cause the microcomputer to stop its current job to be given some information that could occur at any time. The program can be written to find out what caused the interrupt and decide what to do next based on the new information. The program will usually perform some immediate action then continue with what it was doing before the interrupt occurred.

This type of action is most useful for alerting the microcomputer, of events that require action, but that occur on an irregular basis. The program will not need to continually waste instruction time looking to see if the event has taken place.

3.3.3 Status Register Bit Assignments

Most significant bit Least significant bit

7	6	5	4	3	2	1	0
CY/L	OV	SB	SA	F3	F2	F1	IE

3.3.3.1 Status Register Bit Definitions -

BIT DESCRIPTION

7 CARRY/LINK (CY/L)

This bit is set to 1 if a carry occurs from the most significant bit during some machine language instructions.

6 OVERFLOW (OV)

This bit is set if an arithmetic overflow occurs during some machine language instructions.

NOTE

The above two bits may be of no use in an NSC Tiny BASIC program since they are machine code status bits that may be used one of more times during any given BASIC Instruction.

5 SENSE BIT B (SB)

This bit is tied to an external connector pin and may be used to sense external conditions. This is a "read only" bit which is not affected when the contents of the accumulator are copied into the status register by a STAT instruction. It is also the second interrupt input and may be examined by the

"ON" command.

4 SENSE BIT A (SA)

This bit is tied to an external connector pin and may be used to sense external conditions. In addition it acts as the interrupt input when the INTERRUPT ENABLE (status register bit 3) is set. This bit is a "read only" bit. It is the first interrupt input and may be examined by the "ON" command. This bit is used by the NSC Tiny BASIC as the serial input bit from the terminal.

3 USER FLAG 3 (F3)

This bit can be set or reset as a control function for external events or for software status. It is available as an external output from the INS 8073.

2 USER FLAG 2 (F2)

This bit can be set or reset as a control function for external events or for software status. It is available as an external output from the INS 8073. This bit is used by the NSC Tiny BASIC to control the paper tape reader relay.

1 USER FLAG 1 (F1)

This bit can be set or reset as a control function for external events or for software status. It is available as an external output from the INS 8073. This bit is used by the NSC Tiny BASIC as the serial output bit (with inverted data) to the terminal.

Note:

The flag 1, 2 and 3 outputs of the status register serve as latched flags. They are set to the specified state when the contents of the accumulator are copied into the status register. They are in that state until the contents of the status register are modified under program control.

0 INTERRUPT ENABLE FLAG (IE)

The INS 8073 recognizes the interrupt inputs if this flag is set. This bit can be set and reset under program control. When set the NSC Tiny BASIC recognizes external interrupt requests received via the SENSE A or SENSE B inputs . When reset the NSC Tiny BASIC inhibits the INS 8073 from recognizing interrupt requests.

3.4 THE REAL TIME CLOCK/CALENDAR

The MC-1N microcontroller has an on board real time clock that can be very useful when operating as a controller. The clock is a National Semiconductor MM58174 microprocessor bus oriented clock that provides tenths of seconds through months, including leap year calculation. The clock is crystal controlled and includes provisions for a very low drain battery backup.

In the MC-1N the real time clock resides in memory locations #A800 thru #A80F:

ADDRESS	COUNTER	MODE
#A800	Test only	Write only
#A801	Tenths of seconds	Read only
#A802	Units of seconds	Read only
#A803	Tens of seconds	Read only
#A804	Units of minutes	Read or write
#A805	Tens of minutes	Read or write
#A806	Units of hours	Read or write
#A807	Tens of hours	Read or write
#A808	Units of days	Read or write
#A809	Tens of days	Read or write
#A80A	Day of week	Read or write
#A80B	Units of months	Read or write
#A80C	Tens of months	Read or write
#A80D	Years	Read or write
#A80E	Start/Stop	Read or write
#A80F	Interrupt and Status	Read or write

All counters are four bit counters and utilize the lower four bits of the data bus (D0-D3).

3.4.1 Initialization

To have the clock operate predictably, it must be initialized after power up. For normal operation, the clock circuit must not be in the test mode, therefore, set @#A800 = 0. Since the time has not been set yet we should make sure the clock is stopped, therefore set @#A80E = 0. Also, let’s disable the interrupts, therefore, set @#A80F = 0. Once this is done, the clock may now be set.

3.4.2 Setting the clock

To set the clock, load locations #A804 thru #A80D with the desired data. For example to set the clock to 3:58pm on February 12, 1983:

```
@#A804 = 8          The minutes
@#A805 = 5
@#A806 = 5          The hour in 24 hour time (i.e., 3:00pm is 15:00 )
@#A807 = 1
@#A808 = 2          Day of the month
@#A809 = 1
@#A80A = 7          Saturday, the seventh day of the week
@#A80B = 2          The month
@#A80C = 0
@#A80D = 1          Sets leap year correction counter (see below)
```

The things to remember when setting the clock are that the time is kept in 24 hour time so all pm times must have 12 hours added to them and 12 am is the 0 hour. The day of the week is your choice, but convention usually assigns Sunday as the first day of the week. The year setting is really only a leap year correction counter and should be set according to the following:

```
@#A80D = 8          If the year is a leap year
@#A80D = 4          If the year is a leap year plus one year
@#A80D = 2          If the year is a leap year plus two years
@#A80D = 1          If the year is a leap year plus three years
```

If interrupts from the clock are to be utilized, the interrupt jumper must be installed on the MC-1N board (see configuration options below) and your software must have enabled the INS8073's interrupts (see additional I/O above). The MM58174 supports two types of interrupts, single and continuous. These can be set as follows:

```
@#A80F = 0          No interrupt
@#A80F = 1          Single interrupt after 0.5 seconds
@#A80F = 2          Single interrupt after 5.0 seconds
@#A80F = 4          Single interrupt after 60.0 seconds
@#A80F = 9          Continuous interrupts at 0.5 second intervals
@#A80F = 10         Continuous interrupts at 5.0 second intervals
@#A80F = 12         Continuous interrupts at 60.0 second intervals
```

In the single interrupt mode, the interrupt must be respecified each time it is required. In the continuous mode the interrupt will automatically be respecified each time it is read (see below). All interrupt lines are accurate to +/- 16.67ms. Single interrupts will usually be set while the clock is running on an as required basis.

3.4.3 Starting the clock

The clock is started by setting @#A80E = 1. The seconds counters will begin at zero seconds. To stop the clock set @#A80E = 0.

3.4.4 Reading the clock

To read date/time read the counters at locations #A801 thru #A80C. The clock chip updates all time counters ten times per second (every tenth second). If an update had occurred since the last time a counter was read, the chip will output a 15 to let you know that an update has occurred. NSC Tiny BASIC reads memory locations to slow to allow reading the clock times without an update occurring. Also to read in all digits of the date/time will take an elapsed time of more than a second in Tiny BASIC, so use of the clock to do timing control of events faster than a few seconds is not practical unless you write a machine code clock read routine (BASICON has one available in the MC-1N Utility ROM). In any case, if you read a 15 while reading a location, read it again to get the correct digit. A typical Tiny BASIC routine to read a digit might look like:

```
150 B=#A805
155 A=(@B AND 15):IF A=15 A=(@B AND 15):IF A=15 A=(@B AND 15)
```

In the above example the clock counter at #A805 (B) is read into variable A. If A is equal to 15 then the clock is reread. If the second reading of the clock is still 15 then the clock is reread a third time to make sure it really is at 15. This indicates that the clock is not initialized and initial times are not set into the clock. It is possible for any number from 0 to 15 to be present in some of the counters at power up and are initialized only under program control. An example routine to read several digits might look like:

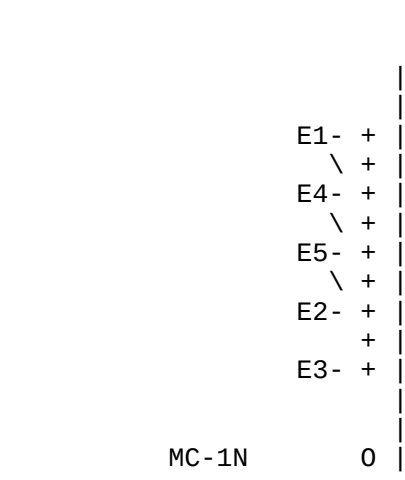
```
200 FOR I=0 TO 3 :  B=#A804+I :  GOSUB500 : @(TOP+I)=A :  NEXTI
                        |
                        |
                        |
500 A=(@B AND 15):IF A=15 A=(@B AND 15):IF A=15 A=(@B AND 15)
510 RETURN
```

The above example will read the minutes and hours and place them into four locations starting at TOP. To insure that an update, that might affect the minutes or hours, has not occurred while reading them, the times should be read at least twice and compared. If they don’t compare, read a third time again and compare with the second read times. This procedure must be continued until two consecutive readings compare.

An illustration may serve to further clarify any misunderstanding. The time registers are read in sequence low order (units) minutes, high order (tens) minutes, low order (units) hours, high order (tens) hours. If on the first reading the time shows 4:59 and the second reading is 4:00 the third reading is 4:00. During the first reading the low order register hours register was updated while reading the minutes register giving a very erroneous time. Other sequences of register readings will require similar checking.

If you are using the interrupt timers in the MM58174, then upon interrupt the first thing you need to do is to read the clock status register. This will reset the interrupt output from the clock chip

3.5 MC-1N OPTIONS AND CONFIGURATION

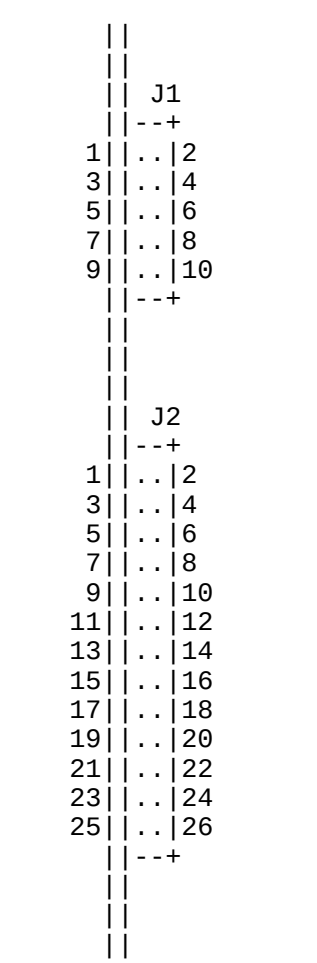


JUMPERS

E1 Clock interrupt jumper
E2 2732 EPROM
E3 2716 EPROM
E4 Baud rate select
E5 Baud rate select

Baud Rate Jumpers

	E4	E5
110	out	out
300	out	in
1200	in	out
4800	in	in



CONNECTOR J1

J1-1	RS232 receive	J1-2	RS232 transmit
J1-3	-5V output	J1-4	Keep-alive input
J1-5	SB input	J1-6	Reset input
J1-7	F3 output	J1-8	F2 output
J1-9	Ground	J1-10	+5v power input

CONNECTOR J2

J1-1	PA3	J2-2	PA2
J1-3	PA1	J2-4	PA0
J1-5	+5v	J2-6	Ground
J1-7	PC7	J2-8	PA4
J1-9	PC6	J2-10	PA5
J1-11	PC5	J2-12	PA6
J1-13	PC4	J2-14	PA7
J1-15	PC1	J2-16	PC0
J1-17	PC2	J2-18	PB7
J1-19	PC3	J2-20	PB6
J1-21	PB0	J2-22	PB5
J1-23	PB1	J2-24	PB4
J1-25	PB2	J2-26	PB3

3.6 APPLICATIONS

- Burglar Alarm or Security System Controller
- Solar Collector System Controller
- Laboratory Experiment Controller
- Interactive Information Display Controller
- Relay Contact Life Tester
- Motor Life Tester
- Battery Discharge Life Tester
- Special Purpose RS-232 Serial to Parallel Converter
- Programmable Battery Backed Timer

APPENDIX A

APPENDIX

A.1 ERROR CODE SUMMARY

A.1.1 NSC Tiny BASIC Error Messages

1. Out of memory
2. Statement used improperly
3. Unexpected character (after illegal statement)
4. Syntax Error
5. Value (format) error
6. Ending quote missing from string
7. GO target line doesn’t exist
8. RETURN without previous GOSUB
9. Expression or FOR-NEXT or DO-UNTIL nested too deeply
10. NEXT without previous matching FOR
11. UNTIL without previous DO
12. Division by zero

A.2 ASCII CODES

ASCII to DECIMAL - HEXADECIMAL - OCTAL CONVERSION TABLE

NUMBER BASE				NUMBER BASE				NUMBER BASE				NUMBER BASE			
10	16	8	CHR	10	16	8	CHR	10	16	8	CHR	10	16	8	CHR
00	00	00	NUL	32	20	40		64	40	100	@	96	60	140	‘
01	01	01	SOH	33	21	41	!	65	41	101	A	97	61	141	a
02	02	02	STX	34	22	42	"	66	42	102	B	98	62	142	b
03	03	03	ETX	35	23	43	#	67	43	103	C	99	63	143	c
04	04	04	EOT	36	24	44	$	68	44	104	D	100	64	144	d
05	05	05	ENQ	37	25	45	%	69	45	105	E	101	65	145	e
06	06	06	ACK	38	26	46	&	70	46	106	F	102	66	146	f
07	07	07	BEL	39	27	47	’	71	47	107	G	103	67	147	g
08	08	10	BS	40	28	50	(	72	48	110	H	104	68	150	h
09	09	11	HT	41	29	51	)	73	49	111	I	105	69	151	i
10	0A	12	LF	42	2A	52	*	74	4A	112	J	106	6A	152	j
11	0B	13	VT	43	2B	53	+	75	4B	113	K	107	6B	153	k
12	0C	14	FF	44	2C	54	,	76	4C	114	L	108	6C	154	l
13	0D	15	CR	45	2D	55	-	77	4D	115	M	109	6D	155	m
14	0E	16	SO	46	2E	56	.	78	4E	116	N	110	6E	156	n
15	0F	17	SI	47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20	DLE	48	30	60	0	80	50	120	P	112	70	160	p
17	11	21	DC1	49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22	DC2	50	32	62	2	82	52	122	R	114	72	162	r
19	13	23	DC3	51	33	63	3	83	53	123	S	115	73	163	s
20	14	24	DC4	52	34	64	4	84	54	124	T	116	74	164	t
21	15	25	NAK	53	35	65	5	85	55	125	U	117	75	165	u
22	16	26	SYN	54	36	66	6	86	56	126	V	118	76	166	v
23	17	27	ETB	55	37	67	7	87	57	127	W	119	77	167	w
24	18	30	CAN	56	38	70	8	88	58	130	X	120	78	170	x
25	19	31	EM	57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32	SUB	58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33	ESC	59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34	FS	60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35	GS	61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36	RS	62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37	VS	63	3F	77	?	95	5F	137	_	127	7F	177	del

10	16	8	CHR	10	16	8	CHR	10	16	8	CHR	10	16	8	CHR
NUMBER BASE				NUMBER BASE				NUMBER BASE				NUMBER BASE			

A.3 LANGUAGE SUMMARY

A.3.1 Command Summary

```
CONT:
LIST [expr]: Lists the current program, with line start option.
NEW expr:    Establishes a new begin of program address.
NEW:         Sets the end of program pointer equal to the begin
             of program pointer.
RUN:         Runs the current program.
```

A.3.2 Statement Summary

CLEAR:	Sets all variables to 0, disables interrupts, and resets all stacks.
DELAY expr:	Loop and do nothing for <expr> units of time. 1-1040 milliseconds, 0=1040 milliseconds.
DO:	Begin of loop statement, may be nested eight deep.
FOR var=expr TO expr [STEP expr]:	FOR loop STEP assumed to be one, may be nested four deep.
GOSUB expr:	Execute subroutine at statement <expr>, may be nested eight deep.
GO[TO] expr:	Jump to program statement <expr>.
IF expr [THEN] statements:	Statements are executed if <expr> is true (non zero).
INPUT $factor:	Put string from terminal into RAM beginning at address <factor>.
INPUT var:	Put value from terminal into <var>.
[LET]$factor=$factor:	Copies one memory location to another.
[LET]$factor="string":	Address <factor> is start of "string" in RAM.
[LET]@factor=expr:	Sets memory location pointed to by <factor> equal to <expr>.
[LET]STAT=expr:	Sets status register to value of <expr>.
[LET]var=expr:	Assigns expression value to <var>.
LINK expr:	Gosub statement for calling microcode subroutines.
NEXT var:	FOR loop termination.
ON <1 or 2> expr:	Interrupt processing statement.
PRINT expr:	Prints the value of <expr>.
PRINT $factor:	Prints the "string" beginning at address <factor>.
PRINT "string":	Prints the "string".
REM [string]:	Remark, for comments.
RETURN:	Termination for GOSUB.
STOP:	Cease program execution.
UNTIL expr:	DO loop termination.

A.3.3 Operator Summary

Arithmetic operators:

addition	+
subtraction	-
multiplication	*
division	/

Relational operators:

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

Logical operators:

AND
OR
NOT

Special Symbols:

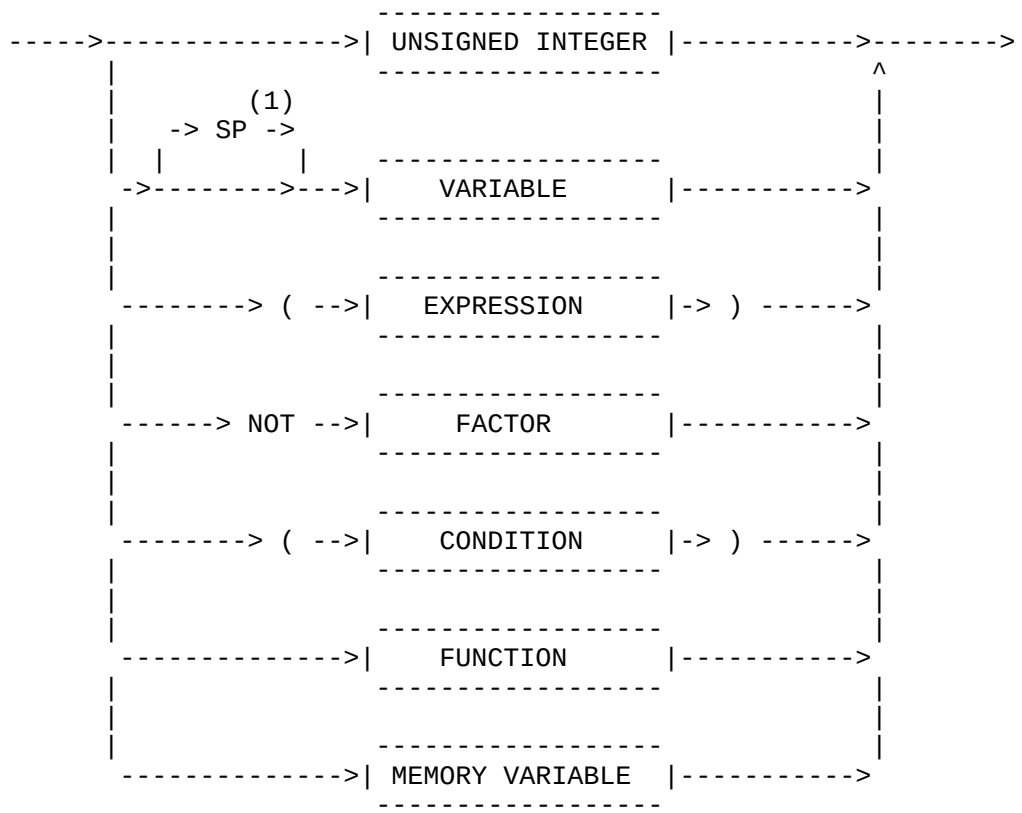
:	Used to separate line statements.
;	Terminates a PRINT statement without a carriage return or line feed.
$	Signifies that the following variable or expression is the address of a string.
@	Signifies that the following variable or expression is the address of a byte.
#	Signifies that the following number is in hexadecimal.
<control C>	Causes program to halt at end of current basic statement.
<control H>	Backspace.
<BREAK>	Halts BASIC program immediately.

A.3.4 Function Summary

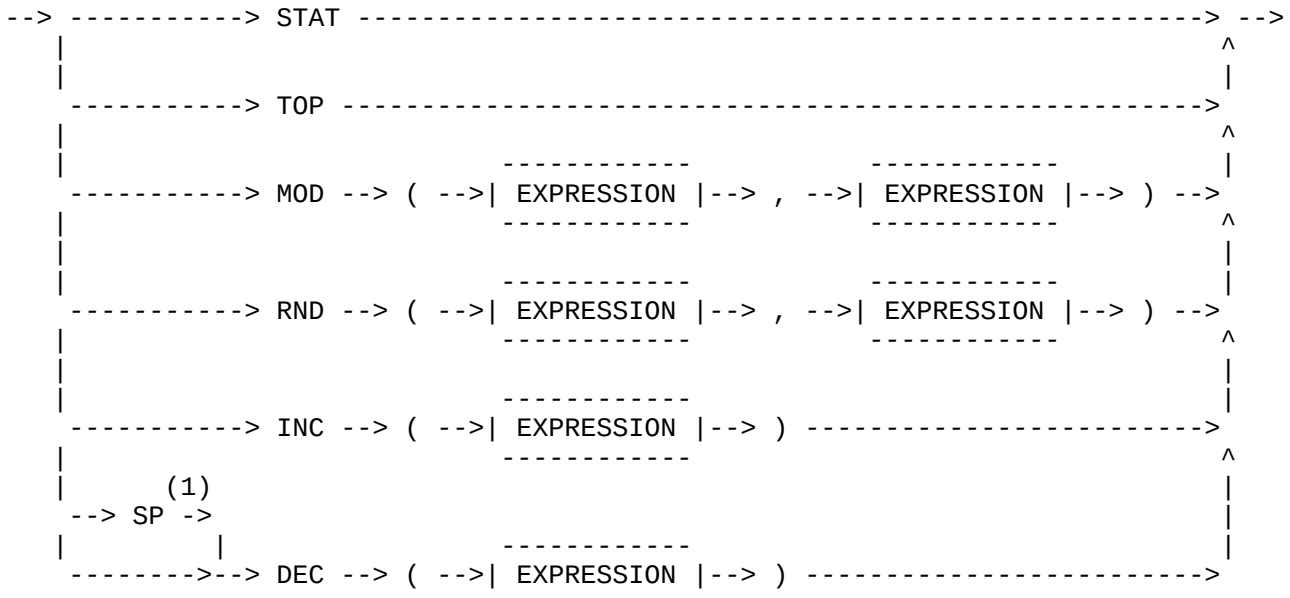
INC (c) DEC (x):	Non-interruptible increment or decrement of a memory location.
MOD(x,y):	Remainder of x divided by y.
RND(x,y):	Random number generator between x and y inclusive.
STAT:	The status register contents.
TOP:	First available RAM memory byte after end of program byte.

A.4 SYNTAX DIAGRAMS (GRAMMAR)

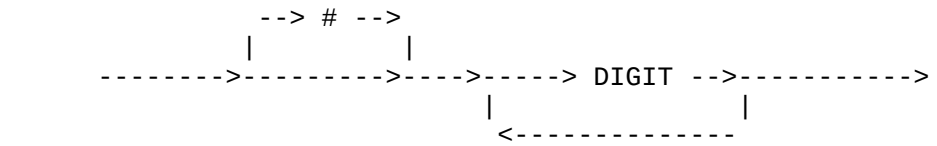
FACTOR



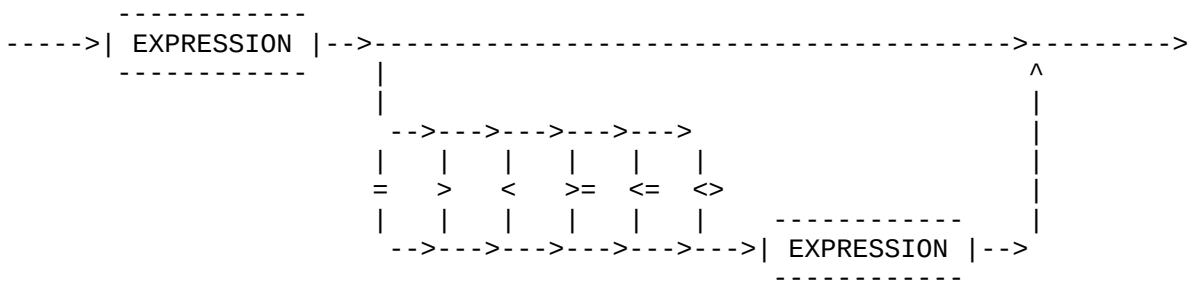
FUNCTION



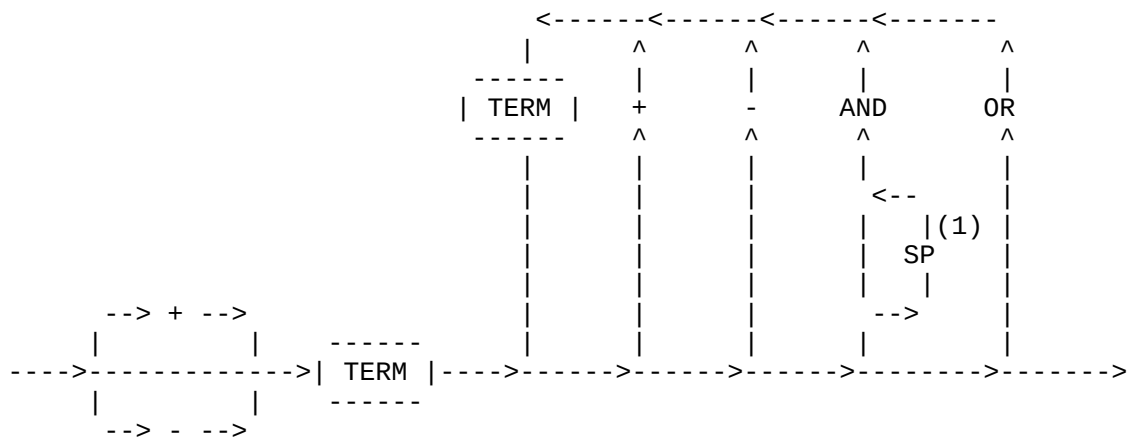
UNSIGNED INTEGER



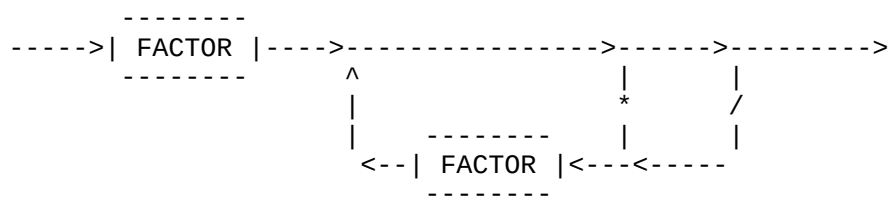
CONDITION



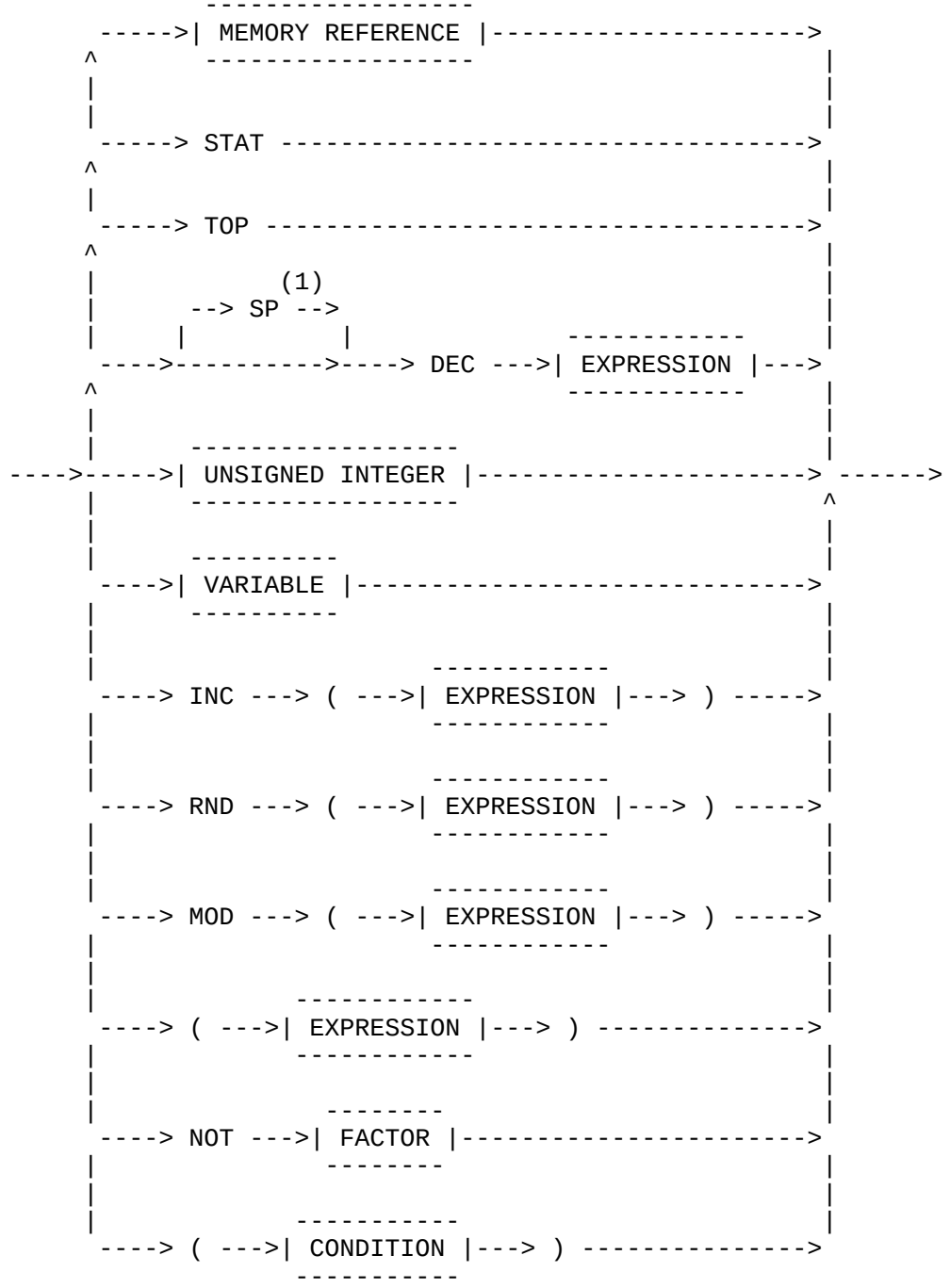
EXPRESSION



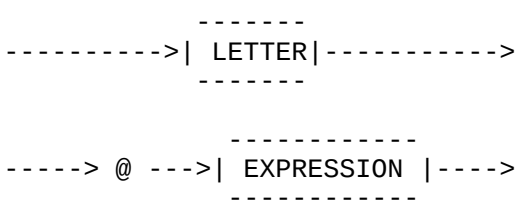
TERM



FACTOR

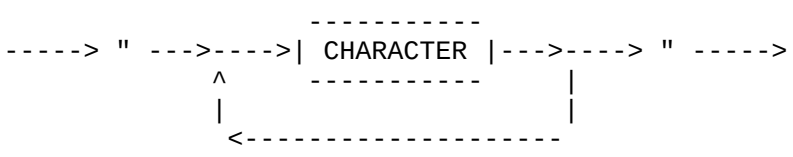

MEMORY REFERENCE
STAT
TOP
(1)
SP
DEC
EXPRESSION
UNSIGNED INTEGER
VARIABLE
INC
RND
MOD
NOT
FACTOR
CONDITION

VARIABLE

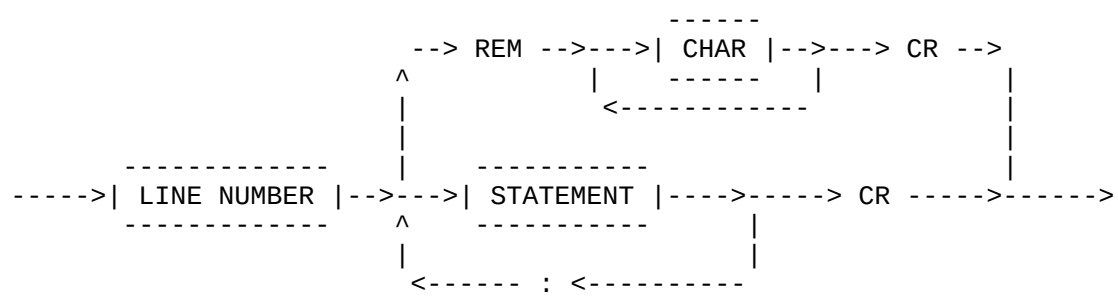


STRING VARIABLE

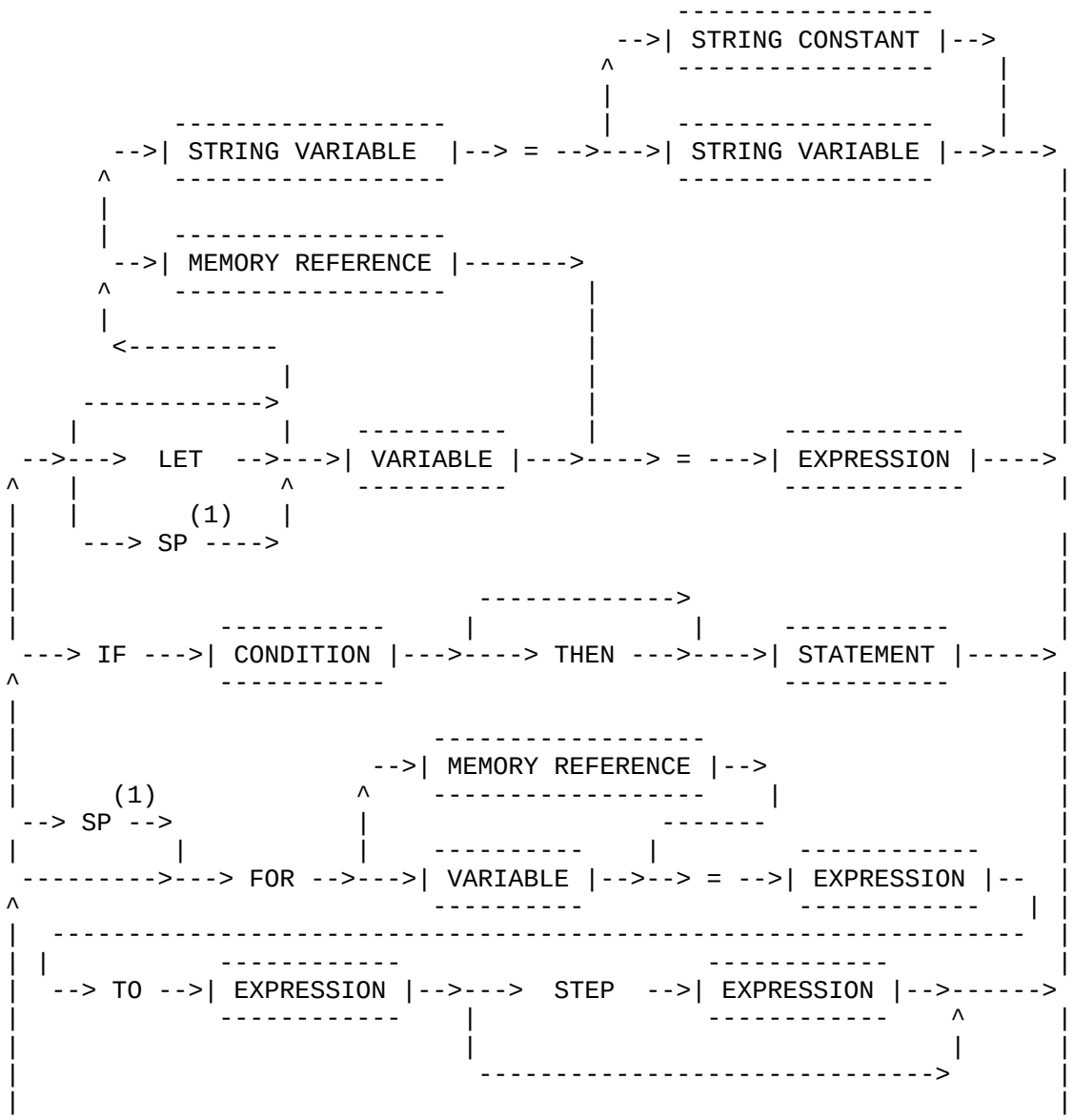
STRING CONSTANT

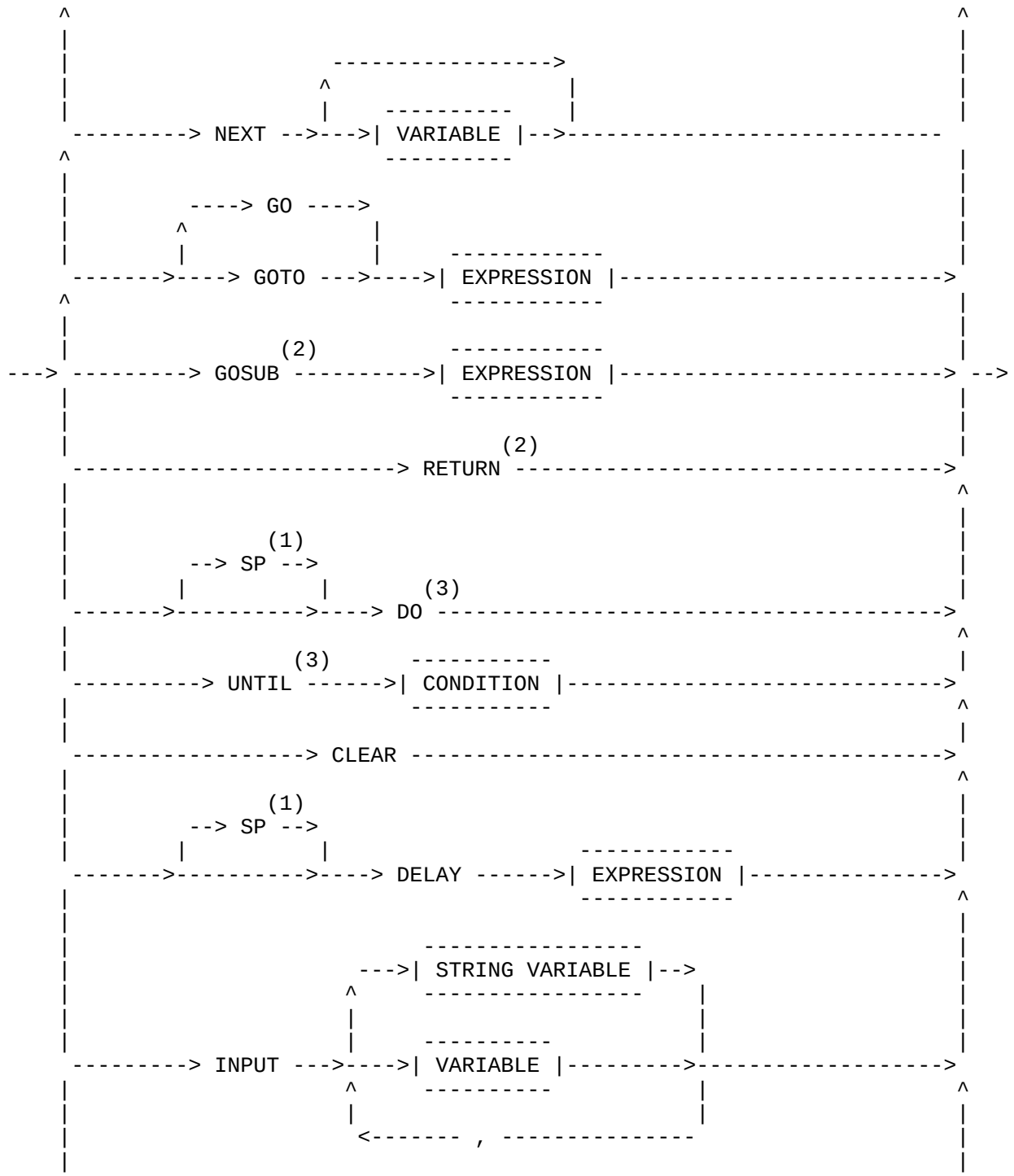


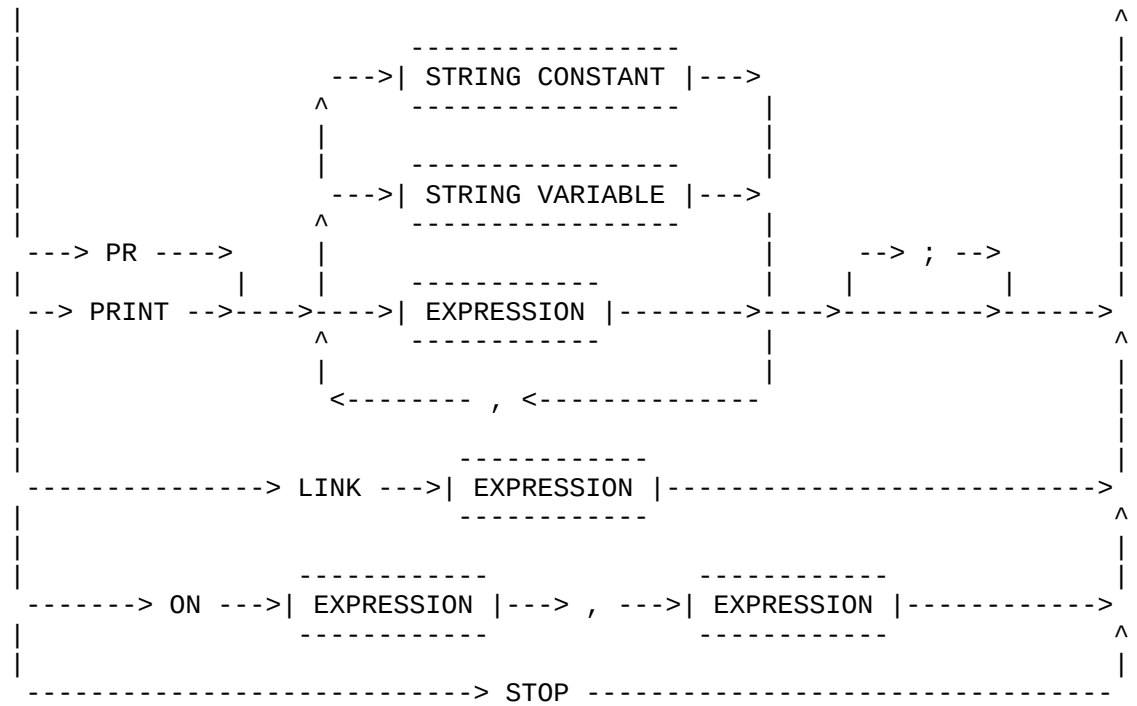
STATEMENT LINE



STATEMENT


STRING CONSTANT
STRING VARIABLE
=
STRING VARIABLE
MEMORY REFERENCE
LET
VARIABLE
=
EXPRESSION
(1)
SP
IF
CONDITION
THEN
STATEMENT
MEMORY REFERENCE
(1)
SP
FOR
VARIABLE
=
EXPRESSION
TO
EXPRESSION
STEP
EXPRESSION


NEXT
VARIABLE
GO
GOTO
EXPRESSION
(2)
GOSUB
EXPRESSION
(2)
RETURN
(1)
SP
(3)
DO
(3)
UNTIL
CONDITION
CLEAR
(1)
SP
DELAY
EXPRESSION
STRING VARIABLE
INPUT
VARIABLE



(1) Normally SPACES are not required except when the first letter a KEYWORD following a HEX number is either "A", "B", "C", "D", "E" or "F".

Example 10

(2) Each GOSUB must have a matching RETURN. (3) Each DO must have a matching UNTIL. (4) Each FOR must have a matching NEXT (STEP is optional).