Artisan Technology Group

Artisan Technology Group is your source for quality new and certified-used/pre-owned equipment

- FAST SHIPPING AND DELIVERY
- TENS OF THOUSANDS OF IN-STOCK ITEMS
- EQUIPMENT DEMOS
- HUNDREDS OF MANUFACTURERS SUPPORTED
- LEASING/MONTHLY RENTALS
- ITAR CERTIFIED SECURE ASSET SOLUTIONS

SERVICE CENTER REPAIRS
Experienced engineers and technicians on staff at our full-service, in-house repair center

WE BUY USED EQUIPMENT
Sell your excess, underutilized, and idle used equipment
We also offer credit for buy-backs and trade-ins
www.artisantg.com/WeBuyEquipment ↗

InstraView™ **REMOTE INSPECTION**
Remotely inspect equipment before purchasing with our interactive website at ***www.instraview.com*** ↗

LOOKING FOR MORE INFORMATION?
Visit us on the web at ***www.artisantg.com*** ↗ for more information on price quotations, drivers, technical specifications, manuals, and documentation

Contact us: (888) 88-SOURCE | sales@artisantg.com | www.artisantg.com

EASON
TECHNOLOGY
We Deliver Productivity

EASON BASIC USERS GUIDE

Revision 1.4
p/n 50-00003-01

Eason Technology, Inc.

7975 Cameron Dr.
Bldg 300
Windsor, CA 95492

Phone (707) 837-0120
FAX (707) 837-2742

Copyright 1995, Eason Technology, Inc. All rights reserved. Specifications subject to change without notice.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

TABLE OF CONTENTS

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CHAPTER 1. EASON TECHNOLOGY BASIC

BASIC is an easy-to-use computer programming language that uses simple commands written in English. Eason Technology 1000 Series Products incorporate BASIC into them to provide a means of controlling the functions built into them: displays, keypads, I/O ports, peripherals, and optional hardware.

Eason BASIC was written to be as similar as possible to industry standards, including popular BASIC implementations used on PC's. Since its creation at Dartmouth College, BASIC has undergone several modifications but still maintains its overall structure. What this means to the user of Eason BASIC, is that programs written for other machines such as an Apple Macintosh[1] or an IBM-PC[2] when converted to Eason BASIC will be usable in general applications with Eason BASIC, but may require some modifications. Generally BASIC statements dealing with a PC's disk I/O or display will have to be re-written, whereas statements dealing with mathematical statements and control flow will not.

Eason BASIC is further enhanced by the utilization of the ApplicationBuilder program. The ApplicationBuilder is a PC program which generates a BASIC program from menu choices and word-processor like screen editing. These programs may be down-loaded to Eason Technology 1000 Series Products and run generally without any errors. Program development times are cut drastically, and extremely high quality code is generated. Contact Eason Technology or your Eason Technology Distributor to obtain a copy of the ApplicationBuilder Diskette.

Please refer to the users guide for the Eason Technology 1000 Series Product you are writing a BASIC program for. Application examples are provided for each product on the ApplicationBuilder diskette. Study these examples and their associated descriptions.

1.1 Eason Technology 1000 Series Products

At the time of printing this manual, two 1000 Series products the Model 1100 and Model 1000 products utilize Eason BASIC. The actual implementation of the command parser, BASIC editor, and I/O structure is identical within each product. The Model 1100 has optional interfaces that may be added to it (counters, A/D's, D/A's, and I/O expanders). These interfaces have specific commands associated with them. On the other hand, the Model 1000 is a hardware subset of the base Model 1100. Chapters 5 through 9 detail information about different interfaces and software options available for Eason Technology products.

[1]. Apple Macintosh is a registered trademark of Apple Computer Corporation.

[2] IBM-PC is a registered trademark of International Business Machine Corporation.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CHAPTER 2. SCREEN EDITOR

This section describes the operation of the 1000 Series Product's built-in screen editor. This editor allows the programmer to edit programs directly on the 1000 Series Product's screen without a PC attached. This feature is most useful for program debug or for constructing and modifying short programs. We recommend that you utilize the ApplicationBuilder to generate more substantial programs. The ApplicationBuilder will shorten your development time, give you much more accurately generated BASIC code, and provide a means of storing, documenting and cataloging your programs.

2.1 Editing Lines During Entry

If an incorrect character is entered as a line is being typed, it can be deleted with the BACKSPACE or DEL keys, or with CTRL-H. After the character is deleted, you can continue to type on the line. Remember to press "Enter" when finished with a line. Otherwise, the line will not be stored in the program.

To delete the entire program currently residing in memory, enter the NEW command. NEW is usually used to clear memory prior to entering a new program (manually). Downloading a new program using the ApplicationBuilder automatically clears the program loaded in memory. If you use another means of downloading (such as a terminal emulator program), you will have to issue a NEW command to clear memory.

2.2 Editing Lines in Completed Programs

After you have entered your program, you may discover that you need to make some changes. To make these modifications, use the LIST statement to display the program lines that are affected.

1. Type the LIST command, or press the F2 key.
2. Type the LIST command followed by the line number, or a range of line numbers, to be edited.

The lines will appear on your screen. The 1000 Series Product has a 48 line buffer. By pressing PgUp, PgDn, UpCursor, or DnCursor you can scroll through this buffer and edit program lines. Remember to press "Enter" when finished with a line. Otherwise, the line will not be stored in the program.

2.2.1 Editing the Information in a Program Line

You can make changes to the information in a line by positioning the cursor where the change is to be made, and by doing one of the following:

Typing over the characters that are already there.
Deleting characters to the left of the cursor, using the BACKSPACE key.
Deleting characters at the cursor position using the DEL key on the number pad.
Inserting characters at the cursor position by pressing the INS key on the number pad. This moves the characters following the cursor to the right making room for the new information.
Adding to or truncating characters at the end of the program line.

If you have changed more than one line, be sure to press "Enter" on each modified line. The modified lines will be stored in the proper numerical sequence, even if the lines are not updated in numerical order.

NOTE:
A PROGRAM LINE WILL NOT ACTUALLY HAVE CHANGES RECORDED WITHIN THE BASIC PROGRAM UNTIL THE ENTER KEY IS PRESSED WITH THE CURSOR POSITIONED SOMEWHERE ON THE EDITED LINE.

You do not have to move the cursor to the end of the line before pressing the Enter key. The BASIC Interpreter remembers where each line ends, and transfers the whole line, even if Enter is pressed while the cursor is located in the middle or at the beginning of the line.

To truncate, or cut off, a line at the current cursor position, type CTRL-T, followed by pressing the Enter key.

2.2.2 Lines Longer Than 40 Characters

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Even though the 1000 Series Product's screen is 40 characters, you may edit lines up to 78 characters long. The 1000 Series Product must distinguish between a line which "continues" on to the next line by placing a horizontal bar "|" at the end of that line and another horizontal bar at the beginning of the next line. The BASIC Interpreter automatically inserts horizontal bars when you type past the end of a line without pressing Enter. Moreover, if you delete or insert characters into a line, the bars are automatically inserted or deleted depending upon whether a line is greater than 39 characters.

If you download lines which are greater than 39 characters in length, the BASIC interpreter will automatically insert bars when the line is displayed on the screen. The bars are only place holders for the display, and do not take up additional memory in the 1000 Series Product's program space.

2.3 Special Keys

The BASIC Interpreter recognizes nine of the numeric keys on the right side of the IBM keyboard. It also recognizes the BACKSPACE key, ESC key, and the CTRL key. The following keys and key sequences have special functions in BASIC.

Key:	**Function**:
BACKSPACE OR CTRL-H	Deletes the last character typed, or deletes the character to the left of the cursor. All characters to the right of the cursor are moved left one position. Subsequent characters and lines within the current logical line are moved up as with the DEL key.
CTRL-C	Returns to the direct mode without saving changes made to the current line. It will also exit auto line-numbering mode.
CURSOR-DOWN	Moves the cursor down one line on the screen.
CURSOR LEFT	Moves the cursor one position left. When the cursor is advanced beyond the left edge of the screen, it will wrap to the right side of the screen on the preceding line.
CURSOR-RIGHT	Moves the cursor one position right. When the cursor is advanced beyond the right edge of the screen, it will wrap to the left side of the screen on the following line.
CURSOR-UP	Moves the cursor up one line on the screen.
DEL	Deletes the character positioned over the cursor. All characters to the right of the one deleted are then moved one position left to fill in where the deletion was made. DEL (delete) is the opposite of INS (insert). Deleting text reduces logical line length.
CTRL-T	Erases from the cursor position to the end of logical line.
END	Moves the cursor to the end of the logical line. Characters typed from this position are added to the line.
CTRL-M or RETURN	Enters a line into the BASIC program. It also moves the cursor to the next logical line.
HOME	Moves the cursor to the beginning of a line. The screen contents are unchanged.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Key:	Function:
INS	Turns the Insert Mode on and off. Insert Mode is indicated by the cursor blotting the entire character position about once a second. When Insert Mode is off, characters typed replace existing characters on the line. The SPACEBAR erases the character at the current cursor position and moves the cursor one character to the right. The CURSOR-RIGHT key moves the cursor one character to the right, but does not delete the character. When Insert Mode is on, characters following the cursor are moved to the right as typed characters are inserted before them at the current cursor position. After each keystroke, the cursor moves one position to the right.
SHIFT-Print Scrn	Sends the current screen contents to the printer, effectively creating a snapshot of the screen.

2.4 Function Keys

Certain keys or combinations of keys let you perform frequently-used commands or functions with a minimum number of keystrokes. These keys are called *function keys.*

The special function keys that appear on the left side or top of your keyboard can be temporarily redefined to meet the programming requirements and specific functions that your program may require.

You will notice that after the 1000 Series Product powers on, these special key functions appear on the bottom line of your screen. These key assignments have been selected for you as some of the most frequently used command.

Initially, the function keys are assigned the following special functions:

BASIC Function Key Assignments

F1	RUN †
F2	LIST
F3	REMOT † (remote on/off *)
F4	CONT †
F5	AUTO
F6	EDIT

* Useful for uploading and downloading programs with terminal programs other than the ApplicationBuilder. See the REMOTE Statement in the EASON BASIC PROGRAMMING GUIDE.

NOTE:
THE † FOLLOWING A FUNCTION INDICATES THAT YOU DO NOT NEED TO PRESS THE RETURN KEY AFTER THE FUNCTION KEY. THE SELECTED COMMAND WILL BE EXECUTED IMMEDIATELY.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

2.5 Help Key

The 1000 Series Product is equipped with a powerful HELP feature. By pressing the HELP key on the 1000 Series Product's front panel or F10 on the IBM keyboard while in the BASIC Interpreter Screen Editor (not running a program), the following screen appears:

```
------- EASON BASIC HELP SYSTEM --------
 COM1 - Display COM port 1 configuration
 COM2 - Display COM port 2 configuration
 I/O  - Display and modify I/O port
 SYS  - Display system parameters
 LITE - Change LCD backlight parameters
 EXIT - Return to Eason Basic
   COM1  COM2   I/O   SYS  LITE  EXIT
```

The HELP screen gives you access to the following functions via keys F1 - F6:

F-Key	LABEL	FUNCTION
F1	COM1	Displays the buffers and status of COM1.
F2	COM2	Displays the buffers and status of COM2.
F3	SYS	Displays system related status.
F4	I/O	Displays 24-bit I/O positions and allows the operator to change them.
F5	LITE	Select LCD backlight auto shutdown, ON, or OFF: *ON* leaves the backlight on at all times. *OFF* leaves the backlight off at all times. *AUTO* turns backlight off after 10 minutes if no activity occurs such as program execution or a key press. The backlight comes back on as soon as a key is pressed.
F6	EXIT	Press this key to leave the HELP system.

The HELP key is treated differently when the BASIC Interpreter is running a program. When running a BASIC program, the HELP key becomes a function key, F10. For example, ON KEY (10) GOSUB 1000 will generate an interrupt subroutine call to location 1000 when the HELP key is pressed. Think of the HELP key as a "pre-labeled" function key, F10. By keeping track of the operational status of your program you can generate context sensitive HELP for your application. For example, by providing a variable named HELP which gets updated whenever the screen contents change, the subroutine which responds to the HELP key can examine this variable and print a specific message out to the user which instructs the user what to do at any given time.

Examine the demo programs supplied with the 1000 Series Product. These programs make use of the function key interrupts, and in particular, the F10 or HELP interrupt.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CHAPTER 3. CONSTANTS, VARIABLES, EXPRESSIONS AND OPERATORS

After you have learned the fundamentals of programming in BASIC, you will find that you will want to write more complex programs. The information in this chapter will help you learn more about the use of constants, variables, expressions, and operators in BASIC, and how they can be used to develop more sophisticated programs.

3.1 Constants

Constants are static values the BASIC Interpreter uses during execution of your program. There are two types of constants: *string* and *numeric.*

A *string constant* is a sequence of 0 to 127 alphanumeric characters enclosed in double quotation marks. The following are sample string constants:

"HELLO"
"$25,000.00"
"Number of Employees"

Numeric constants can be positive or negative. When entering a numeric constant in BASIC, you should not type the commas. For instance, if the number 10,000 were to be entered as a constant, it would be typed as 10000. There are five types of numeric constants: *integer, long integer,* and *floating-point.*

Constant	Description
Integer (%)	Whole numbers between -32768 and +32767. They do not contain decimal points.
Long Integer (&)	Whole numbers between -2147483648 and +2147483647. They do not contain decimal points.
Floating-Point (!)	Positive or negative numbers that include a decimal point. They may or may not be represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally-signed integer or fixed-point number (the mantissa), followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating-point constants is $0.8388607X10^{-19}$ to $-0.8388607X10^{14}$. For example: 235.988E-7 = .0000235988 2359E6 = 2359000000

3.2 Variables

Variables are the names that you have chosen to represent values used in a BASIC program. The value of a variable may be assigned specifically, or may be the result of calculations in your program. If a variable is assigned no value, BASIC assumes the variable's value to be zero.

3.2.1 Variable Names and Declarations

BASIC variable names may be up to 8 characters long. The characters allowed in a variable name are letters and numbers. The first and last character in the variable name must be a letter. Special type declaration characters are also allowed.

Reserved words (all the words used as BASIC commands, statements, functions, and operators) can't be used as variable names. However, if the reserved word is embedded within the variable name, it will be allowed.

Variables may represent either numeric values or strings.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3.2.2 Type Declaration Characters

Type declaration characters indicate what a variable represents. The following type declaration characters are recognized.

Character	Type of Variable
$	String variable
%	Integer variable
&	Long integer variable
!	Floating Point variable

The following are sample variable names for each type.

Variable Type	Sample Name
String variable	N$
Integer variable	LIMIT%
Long integer	A&
Floating point variable	MINIMUM!

The default type for a numeric variable name is floating point. Double-precision (or long), while very accurate, uses more memory space and more calculation time. Floating point is sufficiently accurate for most applications. However, the seventh significant digit (if printed) will not always be accurate. You should be very careful when making conversions between integer, floating point, and double-precision (or long) variables.

Long integers are useful for moderately fast math functions. Many different types of machine controllers require double precision numeric ranges for their input. Long integers were implemented primarily for this purpose.

The following variable is a floating point value by default:

ABC

Variables beginning with FN are assumed to be calls to a user-defined function.

3.2.3 Array Variables

An *array* is a group or table of values referenced by the same variable name. Each element in an array is referenced by an *array variable* that is a subscripted integer or an integer expression. The subscript is enclosed within parentheses. An array variable name has as many subscripts as there are dimensions in the array.

For example,

V(10)

references a value in a one-dimensional array, while

T(1,4)

references a value in a two-dimensional array.

The maximum number of dimensions for an array in BASIC is 16383. Arrays cannot have a size greater than 32767 bytes. e.g. A(8191), b%(16383), and s$(16383) are all valid. A(8192), b%(16384), and s$(16384) are all invalid sizes for arrays. See 3.2.4 for a description of memory space requirements.

NOTE:
All arrays MUST be initialized by a DIM (dimension) statement. If a subscript greater than the maximum specified is used, you will receive the error message "Subscript out of range." The minimum value for a subscript is 1. If a subscript less than 1 is used, you will receive the error message "Subscript error on line xxxx."

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Multi-dimensional arrays (more than one subscript separated by commas) are useful for storing tabular data. For example, an array dimensioned with DIM A(2,5) could be used to represent a two-row, five-column array such as the following:

Column	1	2	3	4	5
Row 1	10	20	30	40	50
Row 2	60	70	80	90	100

In this example, element A(2,3)= 80 and A(1,4)= 40

3.2.4 Memory Space Requirements for Variable Storage

The different types of variables require different amounts of storage depending upon the application. This can be an important consideration.

Variable	Required Bytes of Storage
Integer	2
Long integer	4
Floating point	4

Arrays	Required Bytes of Storage
Integer	2 per element
Long integer	4 per element
Floating point	4 per element

Strings:

Strings require three bytes of overhead, plus the contents of the string as one byte for each string character. Quotation marks marking the beginning and end of each string are not counted.

3.3 Type Conversion

When necessary, BASIC converts a numeric constant from one type of variable to another, according to the following rules:

If a numeric constant of one type is set equal to a numeric variable of a different type, the number is stored as the type declared in the variable name. For example:

```
10  A% = 23.42
20  PRINT A%
RUN
23
```

If a string variable is set equal to a numeric value or vice versa, a "SYNTAX ERROR" occurs.

Logical operators convert their operands to integers and return an integer result. Operands must be with in the range of -32768 to 32767 or an "Overflow" error occurs. When a floating-point value is converted to an integer, the fractional portion is rounded. For example:

```
10 C% = 55.88
20 PRINT C%
RUN
56
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3.4 Expressions and Operators

An expression may be simply a string or numeric constant, a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC are divided into four categories:

Arithmetic
Relational
Logical
Functional

3.4.1 Arithmetic Operators

The following are the arithmetic operators recognized by BASIC. They appear in order of precedence.

Operator	Operation
^	Exponentiation
-	Negation
*	Multiplication
/	Floating-point Division
+	Addition
-	Subtraction

Operations within parentheses are performed first. Inside the parentheses, the usual order of precedence is maintained.

The following are sample algebraic expressions and their BASIC counterparts:

Algebraic Expression	BASIC Expression
$\frac{X-Z}{Y}$	(X-Z)/Y
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
XYZ	X*(Y*Z)
X(-Y)	X*(-Y)

Two consecutive operators must be separated by parentheses.

3.4.1.1 Modulus Arithmetic

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division.

The following are examples of modulus arithmetic.

10.4 MOD 4 = 2
(10/4=2 with a remainder 2)

25.68 MOD 6.99 = 5
(26/7=3 with a remainder 5)

In order of occurrence within BASIC, modulus arithmetic follows integer division. The INT and FIX functions, described in the Chapter 7, are also useful in modulus arithmetic.

3.4.1.2 Overflow and Division by Zero

If, during the evaluation of an expression, a division by zero is encountered, the "DIVISION BY ZERO" error message will be generated (refer to the ON ERROR statement to allow your program to resume execution automatically after an error occurs).

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

If the evaluation of an exponentiation results in zero being raised to a negative power, the "**division by zero**" error message will be generated.

If overflow occurs, the "**overflow**" error message will be generated.

3.4.2 Relational Operators

Relational operators let you compare two values. The result of the comparison is either true (-1) or false (0). This result can then be used to make a decision regarding program flow.

Table 3.1 displays the relational operators.

Table 3.1
Relational Operators

Operator	Relation Tested	Expression
=	Equality	X = Y
<>	Inequality	X <> Y
<	Less than	X < Y
>	Greater than	X > Y
<=	Less than or equal to	X <= Y
>=	Greater than or equal to	X >= Y

The equal sign is also used to assign a value to a variable.

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first:

```
X+Y < (T-1)/Z
```

This expression is true if the value of X plus Y is less than the value of T-1 divided by Z.

3.4.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bit-wise result which is either true (not zero) or false (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

Table 3.2
Results Returned by Logical Operations

Operation	Value	Value	Result
NOT	X	-	NOT X
	T	-	F
	F	-	T
AND	X	Y	X AND Y
	T	T	T
	T	F	F
	F	T	F
	F	F	F
OR	X	Y	X OR Y
	T	T	T
	T	F	T
	F	T	T
	F	F	F
XOR	X	Y	X XOR Y
	T	T	F
	T	F	T
	F	T	T
	F	F	F

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision. For example:

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators convert their operands to 16-bit, signed, two's complement integers within the range of -32768 to +32767. If the operands are not within this range, an error results. If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bits; that is, each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to mask all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to merge two bytes to create a particular binary value. The following examples demonstrate how the logical operators work:

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

3.4.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC has intrinsic functions that reside in the system, such as SQR (square root) or SIN (sine).

BASIC also allows user-defined functions written by the programmer. See the DEF FN statement in Chapter 7.

The CALL instruction allows access to 1000 SERIES PRODUCT machine specific features such as special screen functions or option board functions. The CALL instruction may have optional parameters associated with it. Refer to the CALL instruction in Chapter 7.

3.4.5 String Operators

To compare strings, use the same relational operators used with numbers:

Operator	Meaning
=	Equal to
<>	Unequal
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

The BASIC Interpreter compares strings by taking one character at a time from each string and comparing their ASCII codes. If the ASCII codes in each string are the same, the strings are equal. If the ASCII codes differ, the lower code number will precede the higher code. If the interpreter reaches the end of one string during string comparison, the shorter string is said to be smaller, providing that both strings are the same up to that point. Leading and trailing blanks are significant.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

For example:

"AA"	<	"AB"
"FILENAME"	=	"FILENAME"
"X&"	>	"X#"
"CL"	>	"CL"
"kg"	>	"KG"
"SMYTH"	<	"SMYTHE"
B$	<	"9/12/78" (where B$ = "8/12/78")

String comparisons can also be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

Strings can be concatenated by using the plus (+) sign. For example:

```
10 A$="TO":B$="GET"
20 PRINT A$+B$
30 PRINT A$+B$ + "HER"
RUN
TOGET
TOGETHER
```

3.5 Mathematical Functions

Mathematical functions not intrinsic to BASIC can be calculated as follows:

Function	BASIC Equivalent
Secant	SEC(X)=1/COS(X)
Cosecant	CSC(X)=1/SIN(X)
Cotangent	COT(X)=1/TAN(X)
Inverse Sine	ARCSIN(X)=ATN(X/SQR(-X*X+1))
Inverse Cosine	ARCCOS(X)=ATN(X/SQR(-X*X+1))+ /2
Inverse Secant	ARCSEC(X)=ATN(X/SQR(X*X-1))+SGN(SGN(X)-1* /2
Inverse Cosecant	ARCCSC(X)=ATN(X/SQR(X*X-1))+SGN(X)-1)* /2
Inverse Cotangent	ARCCOT(X)-ATN(X)+ /2
Hyperbolic Sine	SINH(X)-EXP(X)-EXP(-X))/2
Hyperbolic Cosine	COSH(X)=(EXP(X)+EXP(-X))/2
Hyperbolic Tangent	TANH(X)-(EXP(X)-EXP(-X))/(EXP(X)+EXP(-X))
Hyperbolic Secant	SECH(X)=2/(EXP(X)+EXP(-X))
Hyperbolic Cosecant	CSCH(X)=2/(EXP(X)-EXP(-X))
Hyperbolic Cotangent	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
Inverse Hyperbolic Sine	ARCSINH(X)-LOG(X/SQR(X*X+1))
Inverse Hyperbolic Cosine	ARCCOSH(X)=LOG(X+SQR(X*X-1))
Inverse Hyperbolic Tangent	ARCTANH(X)=LOG((1+X)/(1-X))/2
Inverse Hyperbolic Cosecant	ARCCSCH(X)=LOG(SGN(X)*SQR(X*X+1)+1)/X
Inverse Hyperbolic Secant	ARCSECH(X)=LOG(SQR(-X*X+1)+1)/X
Inverse Hyperbolic Cotangent	ARCCOTH(X)=LOG((X+1)/(X-1))/2

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

3.6 Working with Strings

String operations may be combined with other operators. In the next example shows how to create a subroutine that rounds a number, then formats that number to a string complete with leading zeros:

```
- GOTO SCREEN begin
*>SCREEN begin
- PUT LARGE TEXT AT (1,1): "ROUNDING THE NUMBER!"
- PUT TEXT AT (5,3): "(can only round to four places)"
- PUT TEXT AT (3,4): "Enter a number to round:"
- GET NUMBER number! AT (27,4) USING "+####.######" DEFAULT number!
- PUT TEXT AT (6,5): "Decimals of accuracy:"
- GET NUMBER accuracy% AT (27,5) USING "+#" DEFAULT 0
- GET accuracy% RANGE FROM 0 TO 4 OR GOTO begin
  sign%=sgn(number!)
  number!=number*sign%
  temp!=(10^accuracy%)*number!
  wholetmp&=int(temp!)
  test!=temp!-wholetmp&
  if test!<0.5 then goto noadd
  wholetmp&=wholetmp&+1
  label noadd
  number!=wholetmp&/((10^accuracy%))
  number!=number*sign%
- PUT TEXT AT (13,6): "The number is:"
  strnum$=str$(number!)
- COMMENT "now take the number and pad it with the correct number of zeros"
  pad$="0000000000)"
 if len(strnum$)<11 then strnum $=left$(pad$,(11-len(strnum$)))+ strnum$
- PUT TEXT AT (27,6): strnum$
- SOFTKEY (2) "Again" GOTO SCREEN begin
- SOFTKEY (6) "Exit" GOTO SCREEN endscr
- SOFTKEY WAIT
*>SCREEN endscr
  delay 5000
  call gcls()
- END OF PSEUDOCODE
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CHAPTER 4. COMMANDS AND FUNCTIONS

4.1 Introduction

This chapter contains an alphabetically organized description of all commands, statements and functions. *Commands* are defined as statements which can be issued either during the Command Mode, or from a program line. *Statements* are normally executed from a program line rather than from the Command Mode. *Functions* are statements which return a value. For example:

CLEAR is a *command* which sets all numeric variables to zero and all string variables to null, (it also has an optional parameter that specifies the amount of memory set aside for strings during program execution). This command can be issued without a line number while in the Command Mode, or from a program line.

ABS is a *function* which returns the absolute value of an expression. A function is used in an expression and cannot be used by itself.

DATA is a *statement* which is used to store the numeric and string constants that are accessed by the program's READ Statements(s). This statement would not normally be used in the Command Mode.

4.2 Command Summary

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

4.2.1 Counter Interface Command Summary

4.2.2 Analog Interface Command Summary

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

4.2.3 Expanded Digital I/O Interface Command Summary

4.2.4 Extended Memory Option Command Summary

4.2.5 Real Time Clock Command Summary

4.2.6 Event Manager Command Summary

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

ABS Function

Purpose:

To return the absolute value of the expression i.

Syntax:

ABS(*i*)

Comments:

i must be a numeric expression.

Example:

```
PRINT ABS(4*(-6))
 24
Ready
```

Prints 24 as the result of the ABS function.

ASC Function

Purpose:

To return a numeric value that is the ASCII code for the first character of the string z$.

Syntax:

ASC(*z$*)

Comments:

If z$ is null, an **"argument error"** error is returned. If z$ begins with an uppercase letter, the value returned will be within the range of 65 to 90. If z$ begins with a lowercase letter, the range is 97 to 122. Numbers 0 to 9 return 48 to 57, sequentially. See the CHR$ function for ASCII-to-string conversion.

Example:

```
10 A$="TEN"
20 PRINT ASC(A$)
RUN
 84
Ready
```

84 is the ASCII code for the letter *T*.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

ATN Function

Purpose:

To return the arctangent of *x,* wh*ere x* is expressed in radians.

Syntax:

ATN*(x)*

Comments:

The result is within the range of - &/2 to &/2.

The expression *x* may be any numeric type. The evaluation of ATN is performed in single precision unless x is specified as a double precision number (i.e. x& or x#).

Multiply x by &/180 to convert from degrees to radians.

Example:

```
10 PRINT ATN(3)
RUN
 1.249045
Ready
```

Prints the arctangent of 3 radians (1.249045).

AUTO Command

Purpose:

To generate and increment line numbers automatically each time you press the ENTER Key.

Syntax:

AUTO *[line number][,[increment]]*

Comments:

Use AUTO during program entry to eliminate the need for typing line numbers at the beginning of the line.

AUTO begins numbering at *line number* and increments each subsequent line number by *increment.* The default for both values is 10.

The period (.) can be used as a substitute for *line number* to indicate the current line.

If *line number* is followed by a comma, and *increment* is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a *line number* that is already being used, an asterisk appears after the number to warn that any input will replace the existing line. However, pressing RETURN immediately after the asterisk saves the line and generates the next line number.

AUTO is terminated by entering CTRL-C. BASIC will then return to the command level.

NOTE:
The line in which CTRL-C is entered is not saved. To be sure that you save all desired text, use CTRL-C on a blank line.

Examples:

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
AUTO 20,20
```

Generates line numbers 20, 40, 60, and so on.

```
AUTO
```

Generates line numbers 10, 20, 30, 40, and so on.

BIN Function

Purpose:

To return the decimal value of a binary string.

Syntax:

x*=BIN*(v$)

Comments:

v$ contains the binary string to be converted, the range from -2147483647 to +2147483647.

Binary numbers are numbers to the base 2, rather than base 10 (decimal numbers). Conversion is terminated whenever an illegal character is parsed. The result is the evaluation of the string up to the illegal character. If x is negative, 2's (binary) complement form is used. BIN$ is the complement of this function.

Example:

```
PRINT BIN("101100")
 44
Ready
```

BIN$ Function

Purpose:

To return a string that represents the binary value of the numeric argument.

Syntax:

v$*=BIN$*(x)

Comments:

Binary numbers are numbers to the base 2, rather than base 10 (decimal numbers). *x* is rounded to an integer before BIN$*(x)* is evaluated. See the HEX$ and OCT$ functions for hexadecimal and octal conversions. If *x* is negative, 2's (binary) complement form is used.

Example:

```
10 CLS:INPUT "INPUT DECIMAL NUMBER";X
20 A$=BIN$(X)
30 PRINT X;"DECIMAL IS ";A$;" BINARY"
RUN
INPUT DECIMAL NUMBER? 32
32 DECIMAL IS  10000 BINARY
Ready
```

BIT Function

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Purpose:

To return the bit value of a numeric expression at a given position.

Syntax:

***x*=BIT(*n*,*y*)**

Comments:

x is a numeric variable that stores the bit value (0 or 1).

n is a numeric expression which contains the bit position to be examined. It's range is from 0 to 31.

y is a numeric expression where the bit value at given position is evaluated. If y is a floating number or variable, its value will be converted into integer first.

Example:

```
10  A%=17386
20  For i%= 0 to 14
30  Print Bit (i%,A%);
40  Next
RUN
0101011111000010
Ready
```

CALL Statement

Purpose:

To call a 1000 SERIES PRODUCT special built-in subroutine. These subroutines are incorporated to ease the control of special hardware like the LCD display, keypad, or option interfaces.

Syntax:

CALL *subname[(variables)]*

Subroutines:

Subroutines described on the next few pages

CALL BACKLIGHT Statement

Purpose:

To turn the LCD backlight ON, OFF, or place it in automatic mode.

Syntax:

CALL BACKLIGHT *(light)*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

When *light* is set to 1, the backlight will automatically shut off after 10 minutes of inactivity. Inactivity is determined by writing to the screen (text or graphics), or entering data. After the backlight is shut off, entering data from the keypad (or keyboard), or writing to the screen will reactivate the LCD backlight. This is the default mode.

When *light* is set to 2, the backlight will be on all of the time.

When *light* is set to 3, the backlight will be off all of the time.

The backlight parameter is non-volatile. The backlight mode will be remembered by the system at power up, and remain the operating mode until changed.

Example:

```
CALL BACKLIGHT(3)
```

This will turn off the backlight.

CALL BANNER Statement

Purpose:

To write characters on the screen using a double height character font in normal or reverse video.

Syntax:

CALL BANNER *(column, row, v$, scale/color)*

Comments:

column is the x position of the character. *column* may have a range of 1 to 20 if scale = 0; 1 to 10 if scale = 1.

row is the y position of the character. *row* may have a range of 1 to 4 if scale = 0; 1 to 2 if scale = 1.

v$ is a string which is 20 characters or less if scale = 0; 10 characters or less if scale = 1.

scale/color specifies the size and "color" of the graphics characters.

scale/color	Character Size (pixels)	Video
0	11 x 15	Normal
1	22 x 30	Normal
2	11 x 15	Reverse
3	22 x 30	Reverse

Normal character fonts are 5 by 7 pixels. BANNER writes characters which are either 11 by 15 pixels or 22 by 30 pixels. Since BANNER uses graphics to form the characters, it will be significantly slower than writing characters using the PRINT statement. BANNER can write and position either 4 lines of 20 characters or 2 lines of 10 characters.

BANNER restricts the character set to A - Z (upper case only), 0 - 9, +, - , * , / , comma, and period.

By specifying the bottom row (4 if scale = 0, 2 if scale = 1), it is possible to write characters over the softkeys (if they are active). Care must be taken to position the characters around any softkeys present, or to disable softkeys while using row 4 (if scale = 0) or row 2 (if scale = 1).

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

```
CALL BANNER(1,1,"BIG CHARACTERS",0)
```

This statement writes the string "BIG CHARACTERS" starting in the upper left corner of the screen. 11 by 15 pixel normal characters will be generated.

```
CALL BANNER(1,1,"REVERSE CHARACTERS",2)
```

This statement writes the string "REVERSE CHARACTERS" starting in the upper left corner of the screen. 11 by 15 pixel reverse video characters will be generated.

CALL BOX Statement

Purpose:

Draw a box on the LCD display starting with the specified upper right corner, and extending in size to the specified lower right corner.

Syntax:

CALL BOX *(x1,y1,x2,y2,color/fill)*

Comments:

x1 is the upper right corner x position of the box.

y1 is the upper right corner y position of the box.

x2 is the lower left corner x position of the box.

y2 is the lower left corner y position of the box.

color/fill is the characteristic of the box.

color/fill	**Box**
0	Erase Outline Box
1	Draw Outline Box
100	Erase Filled Box
101	Draw Filled Box

Box dimensions are in display pixels. There are 240 pixels in width (numbered 1 through 239), and 64 pixels in height (numbered 1 through 63).

Examples:

```
CALL BOX(20,10,100,50,1)
```

Draws a box starting at pixel location 20,10 (XY location) and extending to 100,50. The pixels are all on.

```
CALL BOX(20,10,100,50,0)
```

This statement erases the previously drawn box.

```
CALL BOX(20,10,100,50,101)
```

This statement draws a filled box with corners at 20,10 and 100,50.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CALL CLRSTK Statement

Purpose:

Empty the Basic stack.

Syntax:

CALL CLRSTK*()*

Comments:

CALL CLRSTK should be used with extreme caution. When used, any active LOOPS or SUBROUTINE RETURN ADDRESSES are lost. This statement should only be used for error handling.

Example:

```
10 START OVER
20 FOR I% = 1 TO 30000
30 PRINT I%
40 IF I% = 2000 THEN GOSUB 100
50 NEXT
60 END
100 CALL CLRSTK()
110 GOTO 20
```

Without CALL CLRSTK(), the stack would overflow after multiple occurrences of loops like lines 20 through 50. CALL CLRSTK() allows the subroutine on line 100 to return to the main program and initialize the stack to zero.

CALL COMLNK Statement

Purpose:

The CALL COMLNK command has been added to ease troubleshooting of serial devices hooked to the 1000 Series. In effect, it turns the Model 1100 and Model 1000 into a rather expensive gender changer. When active, the command takes data coming in one port and sends it out the other port immediately. It is bi-directional, meaning that it doesn't matter which device is on which COM port. This allows you to have a computer (running your favorite terminal emulation or proprietary diagnostic program) on one port of a 1000 Series device and another device on the other COM port. For example, many motion control systems come with a programming package useful for diagnosing problems and setting and checking parameters. Often, the end desire is to have a 1000 Series product communicate with the motion controller. You would typically have a computer connected to COM2 of you Eason device for programming with the ApplicationBuilder and the motion controller connected to COM1. If you wanted to talk directly from your computer to the motion controller, it would mean unplugging and plugging cables until you were blue in the face. Now, you can invoke the CALL COMLNK command. You no longer have to touch a cable change the device that you are communicating to from your computer.

Syntax:

CALL COMLNK(*stopchar,keystop*)

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

stopchar specifies the ASCII character that will break the COMLNK mode. It is a number between 0 and 256 representing the character you wish to terminate the command. If you never intended to use the "space" character, for example, you would use 32 for *stopchar*. Whenever a space occurs on either port, the COMLNK command will be terminated.

keystop specifies if you want the COMLNK mode to be terminated by a key press on the keypad or on a keyboard plugged into the 1000 Series' keyboard port. When *keystop* = 0, a press of a key will not terminate the COMLNK mode. When *keystop* = 1, a key press will immediately break the COMLNK mode.

COMLNK can also be terminated through the 1000 Series interrupt system, with a <Ctrl-C>, or with the "Start Communications" command in the ApplicationBuilder. This is handy if you want to terminate the COMLNK after a specific amount of time, on an input, or on a specific key press. Note: The CONFIG statement can be used to setup the COM ports prior to executing this command.

Examples:

```
10 CALL COMLNK(27,1)
```

This starts the COMLNK mode. With these parameters, the mode can be broken by an <Esc> key (ASCII 27) over either serial port, by the press of a key on the 1000 Series keypad or keyboard, or by any interrupt.

```
10 CALL COMLNK(8,0)
```

This also starts the COMLNK mode. With these parameters, the mode can be broken by a <Backspace> (ASCII 8) over either serial port, or by any interrupt, but not by any key press, unless one is defined with the ON KEY interrupt command. As always, any other interrupt will break the COMLNK mode.

CALL CURSOR Statement

Purpose:

This command allows you to turn the cursor on or off and to set up the size.

Syntax:

CALL CURSOR*(type)*

Comments:

type determines the type of cursor that will be displayed at any given time.

type	**Cursor**
0	Off
1	Underline
2	Block

The cursor type will be maintained in a newly selected state until a screen print operation (using the "print" command), a POS (position cursor) command, or the CALL CURSOR command is issued again. The default type is 1 (Underline).

Examples:

```
10 CALL CURSOR(2)
```

This changes the cursor to a block cursor.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
10 POS(10,2)
20 CALL CURSOR(1)
30 DELAY 1000
40 CALL CURSOR(0)
50 DELAY 1000
60 POS(20,2)
```

This routine positions the cursor at (10,2) as an underline cursor. After 1 second, it turns the cursor off. After another second, the cursor is moved to position (20,2). This automatically turns the underline cursor back on.

CALL GCLS Statement

Purpose:

To clear the graphic screen. This clears lines, boxes, points, and large text that were placed on the screen.

Syntax:

CALL GCLS()

Comments:

Note that the graphic screen is not automatically cleared when a CLS command is issued nor is it automatically cleared when the system returns to the edit mode.

Example:

```
CALL GCLS()
Ready
```

The graphic screen will be cleared.

CALL GETIMAGE Statement

Objective:

This statement will save a screen image or a portion of a screen image into a BASIC array. You can then use CALL PUTIMAGE to quickly place the image anywhere you choose on the 1000 Series display.

Syntax:

CALL GETIMAGE(*xmin*, *ymin* , *xmax*, *ymax*, *arr%(1)*)

Comments:

This statement will store the window defined by ***xmin, ymin, ymax, xmax*** into the array ***arr%*** starting at the first element. Note that ***arr%*** must be defined with a DIM statement prior to executing the CALL GETIMAGE statement.

xmin - left edge of window to save. Range 0 to 239. Must be less than xmax. Note that xmax-xmin must be greater than 5.

xmax - right edge of window to save. Range 0 to 239. Must be greater than xmin. Note that xmax-xmin must be greater than 5.

ymin - top edge of window to save. Range 0 to 63. Must be less than ymax.

ymax - bottom edge of window to save. Range 0 top 63. Must be greater than ymin.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

arr%( ) - the array where the data will be stored. This variable MUST be a short fixed point integer (i.e. must have a % after its name). This variable MUST also be defined with a DIM statement prior to using the CALL GETIMAGE. To calculate the size that the array must be, use the following formula:

```
SIZE = FIX((xmax-xmin*15)/16)*(ymax-ymin)+1
```

Example:

```
10 dim box%(250)
20 call box(1,1,20,20,1)
30 call getimage(0,0,21,21,pbox%(1))
```

This example draws a 20 x 20 pixel square on the display, gets the image of the square and stores it in an array called *pbox*.

CALL HELP Statement

Purpose:

To allow the programmer to open and display text on a screen containing help text without destroying the contents of the present screen.

Syntax:

CALL HELP(*cmd*, "*helptext*")

Comments:

cmd = 0 Clear the background help screen. Use this code to prepare the system for a help message. "*helptext*" is ignored for this command. Example:

To clear the background help screen:

```
CALL HELP (0,"")
```

cmd = 1 through 8 Write a line of help text to the help screen. The *cmd* specifies the line number to be written to. Note that the text does not actually display on screen until a CALL HELP (9,"") is issued. Example:

To write help text to line 1 of the help screen:

```
CALL HELP(1,"Example of HELP TEXT on line 1")
```

cmd = 9 Display help text on the screen and wait for the operator to press a key. The background screen will not be destroyed. Note that data entry will be disrupted. Example:

To display the help text.

```
CALL HELP(9,"")
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

The following example displays a message on the screen until function key HELP (F10) is pressed. Note that the function key can be any of the function keys, F1 through F10. HELP (F10) is normally chosen since it is labeled help.

```
10 ON KEY (10) GOSUB PUTHELP
20 KEY (10),ON
30 CLS
40 PRINT "THIS IS A NORMAL TEXT SCREEN."
50 GOTO 50
60 LABEL PUTHELP
70 CALL HELP (0,"")
80 CALL HELP (1,"Presenting help text on line 1")
90 CALL HELP (5,"Now showing help text on line 5")
100 CALL HELP (8,"Press any key to exit HELP")
110 CALL HELP (9,"")
120 KEY CONT
130 RETURN
```

CALL INVERTIO Statement

Purpose:

Specify a bit pattern which will invert the 24 bit input or output data.

Syntax:

CALL INVERTIO (*v$*)

Comments:

v$ is a string expression, can only consist of 0's and 1's. Specifying a 1 will invert the I/O, a 0 will not. The last digit in the string always refers to bit 0 of the 24 bit I/O. The unspecified bits on the left are assumed to be zeros. Specifying any character other than 0 or 1 will generate unpredictable results.

Example:

```
10 INP A&
20 PRINT A&
30 CALL INVERTIO("1000101110110")
40 INP A&
50 PRINT A&
60 END

RUN

341786
337516
```

CALL LINE Statement

Purpose:

Draw a line beginning and ending at specified points.

Syntax:

CALL LINE *([bx], [by], ex, ey, color)*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

bx is the starting x position of the line in pixels. By omitting this parameter, the previously drawn line's x ending position is used for bx. If no line was previously drawn, 0 will be used instead.

by is the starting y position of the line in pixels. By omitting this parameter, the previously drawn line's y ending position is used for by. If no line was previously drawn, 0 will be used instead.

ex is the ending x position of the line in pixels.

ey is the ending y position of the line in pixels.

color is the characteristic of the box 1 = visible, 0 = invisible (a way to erase a box).

If no beginning point is specified, the line will begin at the last specified end point (from the previously draw line). A line can be displayed by specifying a color of 1 or erased by specifying a color of 0. Line dimensions are in display pixels. There are 240 pixels in width (numbered 0 through 239), and 64 pixels in height (numbered 0 through 63).

Examples:

```
CALL LINE(10,20,100,40,1)
```

Draw a line from pixel location 10,20 to 100,40, turning on the pixels.

```
CALL LINE(10,20,100,40,0)
```

This statement erases the line drawn in the previous example.

```
10 CLS
20 CALL LINE(10,20,50,60,1)
30 CALL LINE(,,100,40,1)
RUN
```

(lines will be drawn)

```
Ready
```

Program line 20 will draw a line from 10,20 to 50,60. Program line 30 will draw a line from 50,60 to 100,40.

CALL LOWBITS Statement

Purpose:

To control the parallel I/O interrupt mode. Three modes are available PATTERN, LEVEL, and CHANGE.

Syntax:

CALL LOWBITS *(mode)*

Comments:

When *mode* is 0 (default mode), the IO24 statement will set up the interrupt for a PATTERN match. The pattern specified by the IO24 statement must be matched as specified prior to an I/O interrupt generation.

When *mode* is 1 , the IO24 statement will set up the interrupt for a LEVEL match. The zeroes specified in the IO24 statement specify the active bits. Any zero specified may cause an interrupt. Re issuing the IO24 statement with a pattern which still has a zero set will generate another interrupt.

When *mode* is 2, the IO24 statement will set up the I/O port for an interrupt when on of the specified bits CHANGEs. Like the LEVEL mode, the zeroes specified in the IO24 statement will specify the active bits. When the specified bit transitions from a low to a high or from a high to a low, an interrupt will occur.

See the IO24 statement.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Examples:

```
10 CALL LOWBITS(2)
20 IO24 "0110"
30 ON IO24 GOSUB 100
40 GOTO 40
100 PRINT "MADE IT"
110 IO24 "0110"
120 RETURN
```

This program demonstrates the use of the CHANGE mode. Line 20 activates I/O bits 0 and 3. When either of them change from a 1 to a 0 or from a 0 to a 1, an interrupt will occur. Line 30 sets up the interrupt to vector to line 100. Line 100 acknowledges the interrupt (for demonstration purposes), and Line 110 re enables the interrupt. Note that all interrupt service routines must end with the RETURN statement.

Line 10 can use CALL LOWBITS(0) or CALL LOWBITS(1). Specifying a CALL LOWBITS(0) would have generated an interrupt only when bits 0 and 3 are both 0. Specifying a CALL LOWBITS(1) would have generated an interrupt whenever bits 0 or 3 are low. This mode can cause multiple interrupts if the interrupt condition (either bits 0 or 3 are still low), is still present when the interrupt is re enabled (line 110).

CALL NFORMAT Statement

Purpose:

Format a number into a string using a "format string".

Syntax:

CALL NFORMAT ***(var,"format string",result string)***

Comments:

var can be any valid numeric value or expression.

format string consists of a string up to 20 characters long. It can be either a constant or a variable. The programmer can select from the following options:

reserves space for a digit.
+ or **-** allows the operator to enter a sign.
. allows the operator to enter a decimal point.
e or **E** allows the operator to enter an exponent

result string is a string variable.

CALL NFORMAT can be used to control the number of digits, lack of or inclusion of exponents, and the sign. Generally, when the desired result is to print a variable in a formatted fashion, PRINT USING is used. When forming strings which will be stored or output to a communications port, CALL NFORMAT can be handy.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

```
NUMB = 123.9888
CALL NFORMAT (NUMB, "###.##",ST$)
PRINT ST$
Ready
RUN
123.99
```

Note that the result is rounded. One feature of NFORMAT when used with fractional numbers, it adds 1 unit to the right of the right most displayed digit. i.e. when NUMB in the example above is .009, ST$ will be "0.01". If you need to keep all of the leading zeros, see the example under 3.6 (Working with Strings).

CALL NINPUT Statement

Purpose:

Perform formatted input for operator entry.

Syntax:

CALL NINPUT ***(a,"format string",response string)***

Comments:

a can be any numeric value or expression.

format string consists of a string up to 20 characters long. It can be either a constant or a variable. The programmer can select from the following options:

reserves space for a digit.
+ or **-** allows the operator to enter a sign.
. allows the operator to enter a decimal point.
e or **E** allows the operator to enter an exponent

response string is a string variable.

The programmer has control over number of digits, sign character, and exponent entry by the operator. A default value can be specified, whereby the operator need only accept the value by pressing enter. The first numeric key that the operator presses will clear the field, and allow a calculator-like data entry. If a sign is allowed, pressing the +/- key will change the sign of the entry. The response is returned as a string.

Note that when the CALL NINPUT function has terminated, the cursor is left one position to the right of the entry area. This is done to prevent accidental scrolling of the display. The user's program must reposition the cursor with the POS or PRINT Statements.

Note: CALL NINPUT **does not** check for valid input data. The users program must check to make sure that the operator has not entered, for example, 100 in a field specified by ##.##.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

Suppose the programmer requires a data entry for 10 digits, signed with no decimal point, and a default value of 1000000.

```
10 P = 1000000
20 CALL NINPUT (P,"+##########",A$)
30 PRINT
40 PRINT A$
RUN
```

1000000	The 1000 SERIES PRODUCT waits for operator input.
1	Operator presses a 1, the entry area is cleared.
123456789	Operator inputs more data.
-123456789	Operator inputs a minus.
	Operator presses ENTER.
-123456789	The result of the PRINT statement on line 40.

CALL NSETUP Statement

Purpose:

Setup formatted input conditions for operator entry.

Syntax:

CALL NSETUP*(a,"format string")*

Comments:

The two parameters above are exactly the same as the first two in CALL NINPUT Statement. NSETUP and NKEY together perform exactly the same function as NINPUT. The difference is while NINPUT cannot be interrupted by software traps such as timers, NSETUP and NKEY can be and as the interrupts finish, program returns to input at current position.

NSETUP is used once to setup the entry conditions. NKEY can then be called multiple times to perform the numeric entry. The next time NSETUP is called, it will setup for the new data entry.

For more information and an example of how to use these two functions, please refer to CALL NKEY Statement on the next page.

CALL NKEY Statement

Purpose:

Receive a key from the standard input and perform formatted input based on that key.

Syntax:

CALL NKEY*(response string,response key)*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

response string is the same as the third parameter in NINPUT statement. *response key* is added here to store the key pressed from the keypad or keyboard.

Before NKEY is called, NSETUP must be issued to setup the data entry environment. Each time NKEY is called, it performs the appropriate action, same as NINPUT, and then exits the routine. As a result, NKEY should be called repeatedly until <ENTER> or other specified keys are pressed.

When an event interrupts while NKEY is being used, the system exits the NKEY routine and service the event. When it returns, the system will service the instruction after the NKEY statement. Therefore, NSETUP followed by NKEY in a loop performs formatted numeric entry that can be interrupted by external events but can always recover.

Note that if NSETUP and NKEY are used, the interrupt subroutines must not use data entry functions at all, which includes NINPUT, NSETUP, NKEY, INPUT, and LINE INPUT. Failure to do so will cause unpredictable results at the data input.

Example:

This uses the example in CALL NINPUT Statement, with the addition of a timer and two function-key interrupts.

```
10 ON KEY(1) GOSUB FKEY1
30 ON KEY(6) GOSUB EXIT
40 ON TIMER(1) GOSUB T1
60 KEY(1),"F1"
80 KEY(6),"EXIT"
90 KEY(1),ON
110 KEY(6),ON
120 TIMER(1),500
130 cls
140 CALL NSETUP(12,"-####.##")
150 CALL NKEY(A$,KY%)
160 POS 1,3:?"A$=";A$;" KEY=";KY%;"    "
170 IF KY%<>13 THEN 150:'Check for the <ENTER> key
180 END
500 LABEL FKEY1
510 KEY STOP:TIMER STOP
520 POS 1,5:PRINT"FUNCTION KEY 1"
530 DELAY 300
540 GOTO OUTSUB
700 LABEL T1
710 KEY STOP:TIMER STOP
720 POS 1,5:PRINT"TIMER 1"
730 DELAY 300:TIMER 1,500
900 LABEL OUTSUB
910 POS 1,5:PRINT"              "
920 KEY CONT:TIMER CONT
930 RETURN
1000 LABEL EXIT
1010 KEY OFF:TIMER OFF
1020 END
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CALL POINT Statement

Purpose:

To return the status of a graphics pixel point on the screen.

Syntax:

CALL POINT *(x, y, status)*

Comments:

x is the x location of the desired display pixel point on the screen.

y is the y location of the desired display pixel point on the screen.

status is a variable which is used to hold the returned value for the status of the pixel point. If the pixel is visible, a 1 is returned. If the pixel is off, a 0 is returned.

There are 240 pixels in width (numbered 0 through 239), and 64 pixels in height (numbered 0 through 63).

Example:

```
10 CALL SET (10,20,1)
20 CALL POINT (10,20,C)
30 IF C = 1 THEN GOTO 60
40 PRINT "THE PIXEL AT 10, 20 IS OFF"
50 END
60 PRINT "THE PIXEL AT 10, 20 IS ON"
70 END
RUN
THE PIXEL AT 10, 20 IS ON
Ready
```

CALL PRTSCRN Statement

Purpose:

To print contents of the screen using an Epson fx compatible printer.

Syntax:

CALL PRTSCRN ( )

Comments:

This only works with the Model 1100.

CALL PUTIMAGE Statement

Objective:

This statement will recall a screen window saved by the CALL GETIMAGE statement.

Syntax:

CALL PUTIMAGE(*xmin*, *ymin*, *arr%(1)*)

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

xmin and xmax will define the upper left corner of a window which will be recalled. Prior to recalling a screen, it must be saved with the CALL GETIMAGE statement. Refer to the CALL GETIMAGE statement for the ranges and descriptions of ***xmin, ymin***, and ***arr%( )***.

Example:

```
10 dim pbox%(250)
20 call box(1,1,20,20,1)
30 call getimage(0,0,21,21,pbox%(1))
40 for n=1 to 150
50 x=rnd(219):y=rnd(43)
60 call putimage(x,y,pbox%(1))
70 next
```

This example draws a 20 x 20 pixel square on the display, gets the image of the square and stores it in an array called *pbox*. It then "puts" a "copy" of the square in 150 random positions on the display.

CALL PULSE Statement

Objective:

Output a fixed number of pulses with an I/O point of Bank 0 (standard I/O). This is useful for motor jogging.

Syntax:

CALL PULSE(*iobit%*, *rampup%*, *totalcnt&*)

Comments:

iobit% is the I/O bit on bank 0 to toggle, rampup% is the number of milliseconds to run at 250 Hz (the remaining pulses run at 500Hz, and totalcnt& is the total number of pulses to move.

Example:

```
call pulse(2,100,32345)
```

This example outputs 32,345 pulses out I/O bit 2 on Bank 0. For 100ms the pulse rate is 250Hz, the rest are output at 500Hz.

CALL READMS Statement

Purpose:

To directly read the contents of the millisecond timer.

Syntax:

CALL READMS(*data&)*

Comments:

data& is a long integer variable (can be floating point) which contains the value of the millisecond timer. The millisecond timer is reset at power on and counts from 0 to 65535. It is incremented every millisecond.

Example:

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
100 CALL READMS(A&)
110 PRINT A&
RUN
 2345
Ready
```

CALL SCANKEY Statement

Purpose:

To return a variable unique to each key on the 1000 SERIES PRODUCT keypad.

Syntax:

CALL SCANKEY *(v%)*

Comments:

v% is the integer variable name which gets assigned the scancode of the currently pressed key.

The key assignments are listed in APPENDIX B.

The SHIFT key does not return a valid argument and is not used with the CALL SCANKEY *(v%)* Command.

CALL SCANKEY can be used to determine the state of any key.

Note: SCANKEY only works with the 1000 SERIES PRODUCT's keypad **not** the IBM compatible keyboard.

Example:

```
10 CALL SCANKEY (JOG%)
20 IF JOG% = 15104 THEN GOSUB JOG
30 GO TO 10
.
.
100 LABEL JOG
110 (send continuous move command to motion controller)
120 CALL SCANKEY (JOG%)
130 IF JOG% = 0 GOTO 150
140 GOTO 120
150 (send stop command)
160 RETURN
```

This example shows how you might use the CALL SCANKEY Command to configure F1 to be a "jog" key. While F1 is pressed, the motor moves. When F1 is released, the motor stops.

CALL SET Statement

Purpose:

To turn a pixel on the LCD screen on or off.

Syntax:

CALL SET *(x, y, color)*

Comments:

x is the x location of the desired display pixel point on the screen.

y is the y location of the desired display pixel point on the screen.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

color turns the pixel on or off. A value of 1 turns the pixel on (makes it visible), while a value of 0 turns it off. There are 240 pixels in width (numbered 0 through 239), and 64 pixels in height (numbered 0 through 63).

Example:

```
10 CLS
20 FOR N=0 TO 99
30 CALL SET (N,25,1)
40 NEXT
50 END
RUN
(line is drawn)
Ready
```

A line starting at pixel 0,25 is drawn to pixel 99,25.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CALL SINPUT Statement

Purpose:

Perform formatted string data input in a fashion which is interruptable by BASIC interrupts.

Syntax:

CALL SINPUT(*string$, key%*)

Comments:

string$ is a dual function input/output string. Its functionality is determined by the value of *key%*. If *key%* is non zero when `CALL SINPUT` is executed, the command will set up a data entry area on screen in accordance to the following rules: If *string$* contains a string full of spaces, the spaces are printed on screen and the cursor is placed on the left most space. If *string$* contains data like "`BROWN `", the string is placed on the screen with a data entry area which is defined by the length of the screen. The cursor is placed to the left of the "N" in "`BROWN`".

Each key that is pressed will cause the `CALL SINPUT` statement to terminate and the key pressed will be returned in *key%* (note that *key%* MUST be a short integer). This allows the program utilizing the `CALL SINPUT` statement to execute a short loop looking for the terminating character.

Once set up, executing `CALL SINPUT` with *key%* zero will restart data entry exactly where it was last exited. This requires special caution if your program uses interrupts, cursor positions must be saved and restored prior to reentry if any printing is performed on-screen.

Examples:

This statement is best explained by the following example:

```
10 CLS
20 KY%=1
30 A$="BROWN      "
40 CALL SINPUT(A$,KY%)
50 IF KY%<>13 THEN KY%=0: GOTO 40
60 PRINT A$
```

Line 20 sets up `KY%` to instruct `CALL SINPUT` to initialize its entry. Line 40 sets up an entry area (in this case in the upper left corner of the screen due to the `CLS` command). It prints: "`BROWN `" on the screen and places the cursor to the right of the "`N`" in "`BROWN`". Any key typed except the left or right arrow keys will clear "`BROWN`" and be placed on the left most (1,1) location. Line 40 will terminate. If the key pressed was not an enter (character 13) in this case, the data entry will continue with `KY%` set to zero and line 40 executed again. This proceeds until an enter key is pressed and `A$` will contain the string formed by all of the printable keys pressed. In any event, the data entry area will not exceed the area defined by line 30.

In this same example, if an enter is the first key pressed, `A$` will contain "`BROWN`" with no trailing spaces. This provides a simple means of providing a "default" data selection.

The `CALL SINPUT` function was intended to operate with the ApplicationBuilder (version 2.14 and later). As such, the maximum flexibility was installed into the command. Using the `GET STRING ... DEFAULT` Pseudocode will invoke this command and utilize its full power.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CALL TERM Statement

Purpose:

To provide a minimal "dumb" terminal operation under the control of a BASIC program.

Syntax:

CALL TERM (*port*, *mode*)

Comments:

port specifies the EASON COM port which the CALL TERM statement will operate on. When ***port***=1 COM1 will be specified, when ***port***=2 COM2 will be specified.

mode specifies the duplex mode under which the CALL TERM statement will operate under. When ***mode***=0 the half duplex will be specified, when ***mode***=1, full duplex will be specified.

The following is a list of control characters, when received from a host computer will activate screen control functions built into the CALL TERM program. Screen functions exist for positioning the cursor, turning it on and off, backspace, carriage return, line feed, clearing the screen and reverse video.

The control characters can be supplemented with parameters which can specify row, column, and length depending upon the control character. Note that parameters have an offset added to them. This is done to keep the parameters from interfering with the XOFF/XON handshaking commands which my occur in the middle of control character sequences. Note that parameters must be received within 50 milliseconds of each other in order to be processed. This was done to insure that the CALL TERM statement will not lock up when sent illegal or missing parameters. The row column cursor positions are defined as follows:

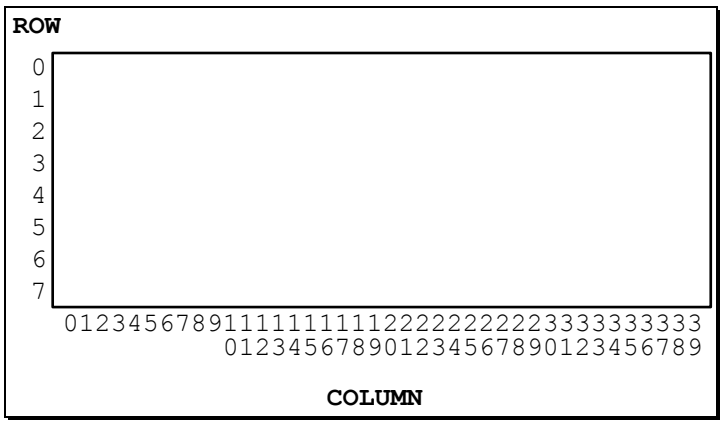

FIGURE 4.1: CALL TERM CHARACTER LOCATION

Each command must be followed by a CARRIAGE RETURN in order to be activated.

EASON BASIC interrupts such as ON KEY, ON IO24, ON TIMER and ON EVENT will terminate the operation of the CALL TERM statement. Note that ON COM will also terminate the operation, but it is not recommended that CALL term and ON COM be set to operate on the same port, unpredictable operation of your system may result.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

COMMANDS (Listed in HEXADECIMAL for clarity):

08 BACKSPACE
Back the cursor column position up 1 position to the left and clear the character at that position. If the cursor is in the first column, it erases that character.

0A LINE FEED
Move the cursor position down 1 row.

0D CARRIAGE RETURN
Move the cursor row position to the beginning of the line (row=0).

11 XON
Resume transmitting characters (from XOFF condition).

13 XOFF
Stop transmitting characters (until XON is received).

17*r c s text* PRINT BANNER CHARACTERS
Print a string of characters at the specified row(r), column (c) and of size(s). Refer to the CALL BANNER statement for a description of the available row, column and sizes associated with this command. Cursor positions are specified by single ASCII characters. The numeric value of these characters has an offset of 32 (decimal). For example to position the cursor at row=0, column =10, size=1, and text = "HELLO", one would send the following characters:

```
'control G' 'space' '*' '!' 'HELLO'\
(In HEX: 17 20 2A 21 48 45 4C 4C 4F)
```

18*rc* SET CURSOR POSITION
Position cursor at the specified row (***r***) and column (***c***). Cursor positions are specified by single ASCII characters. The numeric value of these characters has an offset of 32 (decimal). For example to position the cursor at row=0, column =10, one would send the following characters:

```
'control H' 'space' *'        (In HEX: 18 20 2A)
```

19*n* SET NEXT n CHARACTER TO REVERSE VIDEO
Once the cursor is at the desired position, issuing this command will set the background field to reverse video for ***n*** characters. ***n*** is specified by ASCII codes which have an offset of 32 (decimal). For example if 10 characters are to be selected in reverse video, ***n*** would equal '*' (or 2A in HEX).

1A*n*SET NEXT nn CHARACTERS TO NORMAL VIDEO
Once the cursor is at the desired position, issuing this command will set the background field to normal video for ***n*** characters. ***n*** should not exceed the number of characters available in the current line (40-present row position). ***n*** is specified by ASCII codes which have an offset of 32 (decimal). For example if 11 characters are to be selected in normal video, ***n*** would equal '*' (or 2B in hex). This command is useful to quickly undo command 19 which sets a field to reverse video. One could use command 1F to clear all reverse video attributes, but this will take more time and destroy the reverse video attributes set by previous commands.

1B CURSOR ON
Turn the cursor on.

1C CURSOR OFF
Turn the cursor off.

1D CLEAR SCREEN
Clear the text screen. Does not clear BANNER CHARACTERS or attributes (REVERSE VIDEO).

1E CLEAR TO END OF LINE
Clear from the current cursor position to the end of the current line. If the cursor is placed on the last column in the line, it will clear the last character.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

1F CLEAR ALL REVERSE VIDEO

Clear all reverse video attributes specified by the 19 command. Performing this command may take time >200 msec. Care must be taken not to disrupt your system's operation by this delay. Incoming characters will be suspended if the input COM buffer is full, and keystrokes will be stored in a keyboard buffer.

Example:

The following example invokes the CALL TERM system, and shows a complementary IBM PC QUICK BASIC program to control the terminal.

SYSTEM 1000 PROGRAM:

```
10 CALL TERM(2,0)
RUN
```

IBM PC QUICK BASIC PROGRAM:

```
SETCSR$ = chr$( 24 )
SETRVS$ = chr$( 25 )
SETNRM$ = chr$( 26 )
CSRON$  = chr$( 27 )
CSROFF$ = chr$( 28 )
CLRSCN$ = chr$( 29 )
CLREOL$ = chr$( 30 )
CLRRVS$ = chr$( 31 )
PRINT #1, CLRSCN$;
PRINT #1, SETCSR$;CHR$(32+3);CHR$(32+7);"Imagination*Intelligence=P"
PRINT #1, SETCSR$;CHR$(32+7);CHR$(32+34);SETRVS$;CHR$(32+6);" EXIT ";
```

CHR$ Function

Purpose:

To convert an ASCII code to its equivalent character.

Syntax:

CHR$*(n)*

Comments:

n is a value from 0 to 255.

CHR$ is commonly used to send a special character to the terminal or printer. For example, you could send CHR$(12), a form feed, to the printer.

See the ASC function for ASCII-to-numeric conversion.

Examples:

```
PRINT CHR$(66);
B
Ready
```

This prints the ASCII character code 66, which is the uppercase letter *B*.

```
PRINT CHR$(13);
```

This command prints a carriage return.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CLEAR Command

Purpose:

To set all numeric variables to zero and all string variables to null. CLEAR has the option to reserve the amount of string space available for use by BASIC.

Syntax:

CLEAR *[expression]*

Comments:

expression sets aside string space for BASIC. The default is 512.

BASIC allocates string space dynamically. An **"out of string space"** error occurs only if there is no free memory left to use. Normally this command is used in a program.

The CLEAR command:

Clears all COMMON and user variables
Resets the stack and string space
Disables ON ERROR trapping

NOTE:
Default string space is 512 bytes. Before and after program termination, this value will always be restored. Therefore, string space should be changed only at the beginning of a program.

Examples:

```
CLEAR
```

Zeroes variables and nulls all strings.

```
CLEAR 2048
```

Zeroes variables, nulls strings, allocates 2048 bytes as string space.

CLS Statement

Purpose:

To clear the screen.

Syntax:

CLS

Comments:

The screen may also be cleared by pressing CTRL-HOME.

CLS returns the cursor to the upper-left corner of the screen, and sets the last point referenced to the center of the screen.

Example:

```
CLS
```

This clears the screen, and prints Ready in the upper left corner.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

COM Statement

Purpose:

To enable or disable trapping of communications activity to the specified communications port.

Syntax:

COM*(n)*, ON
COM*(n)*, OFF
COM*(n)*, STOP
COM*(n)*, CONT

Comments:

n is the number of the communications port 1 or 2.

Execute a COM*(n)* ON statement before an ON COM*(n)* statement to allow communications trapping. After executing COM*(n)* ON and ON COM*(n)* statements, BASIC checks continuously for a communications trap before executing a new statement. If a character has been received, a communications trap will be generated (see ON COM*(n)*).

COM*(n)* CONT should be used to restart communications activities after a communications trap has occurred. This will insure that traps which are pending during the communications trap processing will not be lost.

COM*(n)* STOP should be used to suspend communications traps once a COM*(n)* ON has been issued. This will assure that communications traps which occur during the suspension will not be lost. Issuing a COM*(n)* CONT will resume communications trapping. If a COM*(n)* OFF is used, all pending communications traps will be lost.

With COM*(n)* OFF, no trapping takes place, and all communications activity will be lost.

COM(n) to PRN, where n is 1 or 2, redirects the COM port to the printer port. This is usually done when the printer port has already been previously redirected to a COM port.

NOTE:
COM traps are automatically suspended (STOPPED) after they occur. It is the programmer's responsibility to re-enable COM traps after the trap subroutine has been executed. The usual method for re-enabling the traps is to place a CONT statement just prior to the RETURN for the trap subroutine.

CONFIG Statement

Purpose:

Initialize and configure the communications port.

Syntax:

CONFIG *#n,baud,databits,stopbits,parity,handshake,strip,timeout*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

n indicates the communications port to be initialized and configured, 1 for COM1 and 2 for COM2. The parameters are listed as follows:

baud - baud rate, legal values are 3,6,12,24,48,96,192. Each represents a actual baud rate divided by 100. The default value is 96.

databits - number of data bits, either 7 or 8. The default is 8.

stopbits - number of stop bits, 1 or 2. The default is 1.

parity - specifies parity pattern, none (0), even (1) or odd (2). The default is none.

handshake - activate or deactivate handshaking signals (XON/XOFF), use 0 to deactivate and 1 to activate. The default is 1. (XON = CHR$(17) and XOFF = CHR$(19))

strip - strips the line-feed character from the I/O stream if 1, not if 0. The default is 0.

timeout - number of milliseconds that will occur before an input from this port times out. If timeout is 0, the port will not time out. The default is 250.

CONFIG can be issued without all of the parameters. The missing values will take on their default values.

The size of both input COM buffers is 80 bytes.

Examples:

```
CONFIG #1,24,8,1,0,1,1,200
```

This will initialize and configure COM1 to 2400 baud rate, 8 data bits, one stop bit, no parity, handshaking on, strip line-feeds, and time out after 200 milliseconds.

```
CONFIG #2,96,7
```

This will initialize and configure COM2 to 9600 baud rate, 7 data bits, 1 stop bit, no parity, handshaking on, and no stripping and 250 millisecond timeout (default).

CONT Command

Purpose:

To continue program execution after a break.

Syntax:

CONT

Comments:

Resumes program execution after CTRL-C or STOP halts a program. Execution continues at the point where the break happened. If the break took place during an INPUT statement, execution continues after reprinting the prompt.

CONT is useful in debugging, in that it lets you set break points with the STOP statement, modify variables using direct statements, continue program execution, or use GOTO to resume execution at a particular line number.

If a program line is modified, CONT will be invalid and the **"can't continue"** error message will be issued.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

COS Function

Purpose:

To return the cosine of the range of *x*.

Syntax:

COS*(x)*

Comments:

x must be in radians. COS is the trigonometric cosine function. To convert from degrees to radians, multiply by π/180.

Example:

```
10 X=2*COS(.4)
20 PRINT X
RUN
 1.842121
Ready
```

CSRCOL Variable

Purpose:

To return the current column (*x* position) of the cursor.

Syntax:

***x*=CSRCOL**

Comments:

x is a numeric variable receiving the value returned. The value returned is within the range of 1 to 40. CSRCOL is a reserved variable and cannot be assigned a value.

Example:

```
10 CLS
20 POS 12,4
30 PRINT CSRCOL
RUN
(The cursor is positioned to the XY location 12,4)
12  (Printed at the cursor)
Ready
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CSRROW Variable

Purpose:

To return the current row (y position) of the cursor.

Syntax:

***y*=CSRROW**

Comments:

y is a numeric variable receiving the value returned. The value returned is within the range of 1 to 8. CSRROW is a reserved variable and cannot be assigned a value.

Example:

```
10 CLS
20 POS 12,4
30 PRINT CSRCOL
RUN
(The cursor is positioned to the x,y location 12,4)
4  (Printed at the cursor)
Ready
```

DATA Statement

Purpose:

To store the numeric and string constants that are accessed by the program's READ Statements(s).

Syntax:

DATA *constants*

Comments:

Constants are numeric constants in any format (fixed point, floating-point, or integer), separated by commas. No expressions are allowed in the list.

String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

DATA statements are not executable and may be placed anywhere in the program. A DATA statement can contain as many constants as will fit on a line (separated by commas), and any number of DATA statements can be used in a program.

READ statements access the DATA statements in order (by line number). The data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program. The variable type (numeric of string) given in the READ statement must agree with the corresponding constant in the DATA statement, or a **"bad data"** error occurs.

DATA statements may be reread from the beginning by use of the RESTORE statement.

For further information and examples, see the RESTORE statement and the READ statement.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example 1:

```
10 DIM A(10)
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
```

This program segment reads the values from the DATA statements into array A. After execution, the value of A(1) is 3.08, and so on. The DATA statements (lines 110-120) may be placed anywhere in the program; they may even be placed ahead of the READ statement. Notice that the array was dimensioned before use.

Example 2:

```
10 PRINT "CITY";TAB(10);"STATE";TAB(20);"ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,","COLORADO",80211
40 PRINT C$;TAB(10);S$;TAB(20);Z
RUN
CITY  STATE ZIP
DENVER,     COLORADO     80211
Ready
```

This program reads string and numeric data from the DATA statement in line 30.

DEF FN Statement

Purpose:

To define and name a mathematical function written by the user.

Syntax:

DEF FN*name[arguments]expression*

Comments:

name must be a legal variable name. This name, preceded by FN, becomes the name of the function.

arguments consists of those variables in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. Arguments of a user-defined function can only be a numeric variable. Specifying a string or an array element will result in a "**variable required**" error.

expression is an expression that performs the operation of the function. It is limited to one statement.

In the DEF FN statement, arguments serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the argument. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the argument represent, on a one-to-one basis, the argument variables or values that are to be given in the function call.

User-defined functions must be numeric. If a type is specified in the function name and the argument type does not match, a **"syntax error"** error occurs.

A user-defined function may *NOT* be defined more than once in a program.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an **"undefined user function"** error occurs.

Example:

```
.
.
.
400 R=1:S=2
410 DEF FNAB(X,Y)=X^3/Y^2
420 T=FNAB(R,S)
```

Line 410 defines the user-defined function FNAB. The function is called in line 420. When executed, the variable T will contain the value R^3 divided by S^2, or .25.

DELAY Command

Purpose:

To suspend BASIC for specified number of milliseconds.

Syntax:

DELAY *n*

Comments:

n specifies the number of milliseconds to be elapsed before BASIC resumes execution.

n cannot be greater than 65535. If *n* equals 0, there will be no delay.

Example:

```
50 DELAY 1000
```

This will delay program execution for 1 second at line 50.

DELETE Command

Purpose:

To delete program lines or a range of lines.

Syntax:

DELETE *[line number 1][-line number 2]*
DELETE *line number 1-*

Comments:

line number1 is the first line to be deleted.

line number2 is the last line to be deleted.

BASIC always returns to command level after a DELETE command is executed. Unless at least one line number is given, a **"line number required"** error occurs.

Examples:

```
DELETE 20
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Deletes line 20

```
DELETE 30-200
```

Delete lines 30 through 200, inclusively.

```
DELETE -100
```

Deletes all lines up to and including line 100.

```
DELETE 200-
```

Deletes all lines from line 200 to the end of the program.

DIM Statement

Purpose:

To specify the maximum values for array variable subscripts and allocate storage accordingly.

Syntax:

DIM variable*(subscripts)[,variable(subscripts)]...*

Comments:

The maximum number of dimensions for an array is 255.

The minimum value for a subscript is always 1.

If an array variable name is used without a DIM statement, the **"undefined array"** error will appear on the screen. If an array is defined but the subscript is greater than the maximum specified is used, a **"subscript error"** error occurs.

An array, once dimensioned, cannot be redimensioned within the program without first executing a CLEAR or ERASE statement.

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:

```
10 DIM B(10)
20 FOR J=1 TO 10
30 READ B(J)
40 NEXT J
50 DATA 1,2,3,4,5,6,7,8,9,10
RUN
1     2     3     4     5
6     7     8     9     10
Ready
```

This example reads the DATA statements on line 50 and assigns their values to A(1) through A(10), sequentially and inclusively. If the A array is single-precision (default accuracy) then line 10 will allocate 40 bytes of memory to this array (4 bytes times 10 elements).

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

EDIT Command

Purpose:

To display a specified line, and to position the cursor under the first digit of the line number, so that the line may be edited.

Syntax:

EDIT *line number*
EDIT.

Comments:

line number is the number of a line existing in the program. A period (.) refers to the current line. The following command enter EDIT at the current line: **EDIT .** When a line is entered, it becomes the current line. The current line is always the last line referenced by an EDIT statement, LIST command, or error message. If *line number* refers to a line that does not exist in the program, an **"undefined line number"** error occurs.

Example:

```
EDIT 260
```

Displays program line number 260 for editing.

END Statement

Purpose:

To terminate program execution and returns the system to the command level.

Syntax:

END

Comments:

END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a **"break in line *xxxx"*** message to be printed. An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

Example:

```
400 IF J>2000 THEN END ELSE 10
```

Ends the program and returns to command level when the value of J exceeds 2000, otherwise, the program continues on line 10.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

ERASE Statement

Purpose:

To eliminate arrays from a program.

Syntax:

ERASE *list of array variables*

Comments:

Arrays may be redimensioned after they are erased, or the memory space previously allocated to the array may be used for other purposes.

If an attempt is made to redimension an array without first erasing it, an error occurs.

Example:

```
100 DIM X (100)
110 DIM Y (120)
450 ERASE X,Y
460 DIM X(3,4)
```

Arrays X and Y are eliminated from the program. The X array is redimensioned to a 3-column by 4-row array (12 elements), all of which are set to a zero value.

ERR and ERL Variables

Purpose:

To return the error code (ERR) and line number (ERL) associated with an error.

Syntax:

v=ERR
v=ERL

Comments:

The variable ERR contains the error code for the last occurrence of an error.

The variable ERL contains the line number of the line in which the error was detected.

The ERR and ERL Variables are usually used in IF-THEN, or ON ERROR...GOTO, or GOSUB statements to direct program flow in error trapping.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct mode statement, use a line of the following form:

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

Otherwise, use the following:

```
10 IF ERR=error code THEN...GOSUB 4000
20 IF ERL=line number THEN...GOSUB 4010
```

NOTE:
If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM.

Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in an assignment statement.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

ERROR Statement

Purpose:

To simulate the occurrence of an error, or to allow the user to define error codes.

Syntax:

ERROR *integer expression*

Comments:

The value of *integer expression* must be greater than 0 and less than 255.

If the value of *integer expression* equals an error code already in use by BASIC, the ERROR statement simulates the occurrence of that error, and the corresponding error message is printed.

A user-defined error code must use a value greater than any used by the BASIC error codes. There are 50 BASIC error codes at present. See APPENDIX B: ERROR CODES for a listing of BASIC ERROR CODES. It is preferable to use a code number high enough to remain valid when more error codes are added to BASIC.

User-defined error codes may be used in an error-trapping routine.

If an ERROR statement specifies a code for which no error message has been defined, BASIC responds with the message **"illegal error code."**

Execution of an ERROR statement for which there is no error-trapping routine causes an error message to be printed and execution to halt.

Examples:

The following examples simulate error 15 (the code for **"string too long"**):

```
10 S=4
20 T=5
30 ERROR S+T
40 END
Ready
RUN
string overflow in 30
```

Or, in direct mode:

```
Ready
ERROR 9
string overflow
Ready
```

The following example includes a user-defined error code message.

```
.
.
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B>5000 THEN ERROR 210
.
.
400 IF ERR=210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL=130 THEN RESUME 120
.
.
```

EXP Function

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Purpose:

To return *e* (the base of natural logarithms) to the power of *x*.

Syntax:

EXP*(x)*

Comments:

x must be less than 43.57330

If EXP overflows, the **"overflow in exp"** error message appears and program is halted.

EXP*(x)* is calculated in single-precision.

Example:

```
10 X = 5
20 PRINT EXP(X-1)
RUN
54.59815
Ready
```

Prints the value of *e* to the 4th power.

FIX Function

Purpose:

To truncate *x* to a whole number.

Syntax:

FIX(x)

Comments:

FIX does not round off numbers, it simply eliminates the decimal point and all characters to the right of the decimal point.

FIX*(x)* is equivalent to SGN*(x)**INT(ABS*(x)).* The major difference between FIX and INT is that FIX does not return the next lower number for negative *x*.

FIX is useful in modulus arithmetic.

Examples:

```
PRINT FIX(58.75)
58
Ready
PRINT FIX(-58.75)
-58
Ready
```

FOR and NEXT Statements

Purpose:

To execute a series of instructions a specified number of times in a loop.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Syntax:

FOR *variable* = *x* TO *y* *[*STEP *z]*

.

.

.

NEXT*[variable]*

Comments:

variable is used as a counter and can only be a numeric variable. Specifying a string or an array element will result in a "**numeric variable required**" error.

x, y, and z are numeric expressions. Specifying a string expression will result in a "**syntax**" error.

STEP *z* specified the counter increment for each loop.

The first numeric expression *(x)* is the initial value of the counter. The second numeric expression *(y)* is the final value of the counter.

Program lines following the FOR statement are executed until the NEXT statement is encountered. Then, the counter is incremented by the amount specified by STEP.

If STEP is not specified, the increment is assumed to be 1.

A check is performed to see if the value of the counter is now greater than the final value *(y)*. If it is not greater, BASIC branches back to the statement after the FOR statement, and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR-NEXT loop.

The body of the loop is always executed at least once, even if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

Nested Loops:

FOR-NEXT loops may be nested; that is, a FOR-NEXT loop may be placed within the context of another FOR-NEXT loop. When loops are nested, each loop must have a unique variable name as its counter.

The NEXT statement for the inside loop must appear before that for the outside loop.

The *variable* in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement.

If a NEXT statement is encountered before its corresponding FOR statement, a **"next without for"** error message is issued and execution is terminated.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

A Warning About Breaking Out of FOR-NEXT Loops:

FOR-NEXT loops use the BASIC stack to "remember" where the top of the loop is. The FOR statement saves its line number on the stack so the NEXT statement can figure out where to branch to if the ending conditions have not been met. For this reason, branching out of the FOR-NEXT loop's body will leave the FOR statement's line number on the stack. This can create problems if this process is done repeatedly. Even worse, it can create a problem which normal testing would not catch, and only after months of operation will the program fail.

The indication of failure is the OUT OF MEMORY error. To avoid this failure, do not break out of a FOR-NEXT loop. If the program must break out (or an error, or some other non-normal event), use a CALL CLRSTACK statement in the main body of the program to keep the stack from filling up.

Examples:

The following example prints integer values of the variable I% from 1 to 10 in steps of *z*. For fastest execution, I is declared as an integer by the % sign.

```
10 K=10
20 FOR I%=1 TO K STEP 2
30 PRINT I%
.
.
.
60 NEXT
RUN
 1
 3
 5
 7
 9
Ready
```

In the next example, the loop executes 10 times. The final value for the loop variable is always set before the initial value is set.

```
10 S=5
20 FOR S=1 TO S+5
30 PRINT S;
40 NEXT
RUN
1  2  3  4  5  6  7  8  9  10
Ready
```

FRE Function

Purpose:

To return system parameters.

Syntax:

FRE*(x)*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

x specifies the desired system parameter (0 - 9, see below).

Before FRE(*x*) returns the desired system parameter, BASIC initiates a "garbage collection" activity. Data in string memory space is collected and reorganized, and unused portions of fragmented strings are discarded to make room for new input.

If FRE is not used, BASIC initiates an automatic garbage collection activity when all string memory space is used up. BASIC will not initiate garbage collection until all free memory has been used. Garbage collection may be instantaneous or may take a few seconds.

FRE(0) or any string forces a garbage collection before returning the desired system parameter. Therefore, using FRE(0) periodically will result in shorter delays for each garbage collection. FRE(0) will always return a 0.

The system parameter returned for each value of x is:

x	*Returns*
0	0
1	program size
2	maximum string space, default is 512
3	available string space
4	space taken by user-defined functions
5	space taken by arrays
6	space taken by variables
7	available memory
8	stack space used
9	memory option, 32 or 64

It should be noted that the CTRL-C function cannot be used during this housecleaning process.

Example:

```
PRINT FRE(0)
14542
Ready
```

GOSUB...RETURN Statement

Purpose:

To branch to, and return from, a subroutine.

Syntax:

GOSUB *line number* or *label* or *screen name*

.

.

.

RETURN *[line number* or *label* or *screen name]*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

line number or label or screen name is the first line number of or label or screen name for the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

A RETURN statement in a subroutine causes BASIC to return to the statement following the most recent GOSUB statement. A subroutine can contain more than one RETURN statement, should logic dictate a RETURN at different points in the subroutine.

Subroutines can appear anywhere in the program, but must be readily distinguishable from the main program (use a LABEL statement to identify the subroutine).

To prevent inadvertent entry, precede the subroutine by a STOP, END, or GOTO statement to direct program control around the subroutine.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE"
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ready
```

The END statement in line 30 prevents re-execution of the subroutine.

GOTO Statement

Purpose:

To branch unconditionally out of the normal program sequence to a specified line number or label or screen.

Syntax:

GOTO *line number* or *label* or *screen name*

Comments:

line number (or label) is any valid line number (or label) within the program.

If *line number* is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after *line number*.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

```
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
RUN
R = 5  AREA = 78.4996
R = 7  AREA = 153.8599
R = 12 AREA = 452.1598
Out of data in 10
Ready
```

The **"out of data"** advisory is generated when the program attempts to read a fourth DATA statement (which does not exist) in line 60.

HEX Function

Purpose:

To return the decimal value of the hexadecimal string.

Syntax:

***x* = HEX(*v$*)**

Comments:

v$ contains the hexadecimal string to be converted, the range from -2147483647 to +2147483647. Hexadecimal numbers are numbers to the base 16, rather than base 10 (decimal numbers). Conversion is terminated whenever an illegal character is parsed. The result is the evaluation of the string up to the illegal character. If *x* is negative, 2's (binary) complement form is used. HEX$ is the complement of this function.

Example:

```
PRINT HEX( "FFFF" )
 65535
Ready
```

HEX$ Function

Purpose:

To return a string that represents the hexadecimal value of the numeric argument.

Syntax:

***v$*=HEX$(*x*)**

Comments:

Hexadecimal numbers are numbers to the base 16, rather than base 10 (decimal numbers). *x* is rounded to an integer before HEX$(*x*) is evaluated. See the BIN$ and OCT$ functions for binary and octal conversions. If *x* is negative, 2's (binary) complement form is used.

Example:

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
10 CLS:INPUT "INPUT DECIMAL NUMBER";X
20 A$=HEX$(X)
30 PRINT X;"DECIMAL IS "A$";HEXADECIMAL"
RUN
INPUT DECIMAL NUMBER? 32
32 DECIMAL IS 20 HEXADECIMAL
Ready
```

IF Statement

Purpose:

To make a decision regarding program flow based on the result returned by an expression.

Syntax:

IF *expression[,]* THEN *statement(s)* *[*ELSE *statement(s)]*

Comments:

If the result of *expression* is non zero (logical true), the THEN part of the statement is executed.

If the result of *expression* is zero (false), the THEN or GOTO line number or label or screen name is ignored and the ELSE line number or label or screen name, if present, is executed. Otherwise, execution continues with the next executable statement.

The *statement* may be a GOTO *label* or *screen name*. It cannot be a DELAY or GOSUB statement.

Because IF...THEN...ELSE is all one statement, the ELSE clause cannot be on a separate line. All must be on one line.

Testing Equality

When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, test against the range over which the accuracy of the value may vary.

For example, to test a computed variable A against the value 1.0, use the following statement:

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

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example:

In the following example, a test determines if N is greater than 10 and less than 20. If N is within this range, DB is calculated and execution branches to line 300. If N is not within this range, execution continues with line 110.

```
100 IF(N<20) and (N>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
```

The next statement causes printed output to go either to the terminal or to the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the terminal.

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

INKEY Function

Purpose:

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

To return a key read from the keyboard or COM port by returning its ASCII code or the extended scancode as an integer.

Syntax:

v%*=INKEY*([#n])

Comments:

n indicates the device number, 1 for COM1 and 2 for COM2, and the keyboard is used for any other arguments.

See Appendix B for a list of key codes.

The normal ASCII character is returned to lower byte in *v%*. For extended code, the scancode is put in high byte and low byte contains a zero. For example, A returns 65, while up arrow returns 18,432 (4800 hexadecimal).

If no key is pending in the keyboard buffer, zero is returned.

If several characters are pending, only the first is returned.

No characters entered from the keyboard or received on the serial ports are displayed on the screen, and all keystrokes are passed to the program. In contrast to the INKEY$ function, INKEY will return all the keyboard codes, including the function keys, etc. The extended keys are returned with the extended code multiplied by 256.

Example:

```
10 CLS
20 POS 1,1:PRINT "WAITING FOR KEY..."
30 A% = INKEY()
40 IF A%<>0 THEN GOSUB 100
50 GOTO 20
100 POS 1,3:PRINT "     "
110 POS 1,3:PRINT HEX$(A%)
110 RETURN
```

This program will wait for the keystrokes from the keyboard and echo each key's ASCII or extended code on the screen at row 3, column 1 in hexadecimal.

Since an INKEY statement scans the keyboard only once, place INKEY statements within loops to provide adequate response times for the operator.

INKEY$ Function

Purpose:

To return one character read from the keyboard or COM port.

Syntax:

v$*=INKEY$*([#n])

Comments:

n indicates the device number, 1 for COM1 and 2 for COM2, and the keyboard is used for any other arguments.

See Appendix B for a list of key codes.

If no character is pending in the keyboard or COM port buffer, a null string (length zero) is returned.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

If several characters are pending, only the first is returned.

No characters are displayed on the screen, and all valid ASCII characters are passed to the program.

Example:

```
10 CLS: PRINT"PRESS RETURN"
20 TLIMIT% = 1000
30 GOSUB 1010
40 IF TOUT% THEN PRINT "TOO LONG" ELSE PRINT "GOOD SHOW"
50 PRINT RESP$
60 END
1000 REM TIMED INPUT SUBROUTINE
1010 RESP$=""
1020 FOR N%=1 TO TLIMIT%
1030 A$=INKEY$():IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TOUT%=0:RETURN
1050 RESP$=RESP$+A$
1060 NEXT N%
1070 TOUT%=1:RETURN
```

When this program is executed, and if the RETURN key is pressed before 1000 loops are completed, then "GOOD SHOW" is printed on the screen. Otherwise, "TOO LONG" is printed.

Since an INKEY$ statement scans the keyboard only once, place INKEY$ statements within loops to provide adequate response times for the operator.

INP Statement

Purpose:

To return a 24-bit integer read from the parallel port into a long integer variable.

Syntax:

INP *variable&*

Comments:

variable& represents a valid long integer variable.

INP is the complementary function to the OUT statement.

The 1000 SERIES PRODUCT's 24 bit I/O port is a flexible input / output port. All 24 bits default to an "input" state on power up. This is achieved by writing a logic 1 to the port. Any line can be output to by issuing an OUT statement or one of it's variants. If a bit is written to a 0 by the OUT instruction, it no longer can be used as an input, since an external device cannot pull it high. It still can be read, but its value will be a 0 until it is written to a 1 and allowed to float high.

The highest byte of A& will always be 0.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

```
100 INP A&
```

Upon execution, variable A& will receive the value present on the 24-pin parallel port. The number returned will be within the range of 0 to FFFFFF hexadecimal (16777215, decimal).

INPUT Statement

Purpose:

To prepare the program for input from the console during program execution.

Syntax:

INPUT ***[prompt string;] list of variables***
INPUT ***[prompt string,] list of variables***

Comments:

prompt string is a request for data to be supplied during program execution.

list of variables contains the variable(s) that stores the data in the prompt string.

Each data item in the *prompt string* must be surrounded by double quotation marks, followed by a semicolon or comma and the name of the variable to which it will be assigned. If more than one *variable* is given, data items must be separated by commas.

The data entered is assigned to the variable list. The number of data items supplied must be the same as the number of variables in the list.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item input must agree with the type specified by the variable name.

Too many or too few data items, or the wrong type of values (for example, numeric instead of string), causes the message **"bad data redo"** to be printed.

A comma may be used instead of a semicolon after *prompt string* to suppress the question mark. For example, the following line prints the prompt with no question mark:

INPUT "ENTER BIRTHDATE",B$

When an INPUT statement is encountered during program execution, the program halts, the prompt string is displayed, and the operator types in the requested data. Strings that input to an INPUT statement need not be surrounded by double quotation marks unless they contain commas or leading or trailing blanks.

When the operator presses the RETURN key, program execution continues.

INPUT and LINE INPUT statements have built-in PRINT statements. When an INPUT statement with a quoted string is encountered during program execution, the quoted string is printed automatically (see the PRINT statement).

The principal difference between the INPUT and LINE INPUT statements is that LINE INPUT accepts special characters (such as commas) within a string, without requiring double quotation marks, while the INPUT statement requires double quotation marks.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example 1:

To find the square of a number:

```
10 INPUT X%
20 PRINT X%;" SQUARED IS ";X%^2
30 END
RUN
?
```

The operator types a number (5) in response to the question mark:

```
5 SQUARED IS 25
Ready
```

Example 2:

To find the area of a circle when the radius is known:

```
10 INPUT "WHAT IS THE RADIUS";R
20 A=PI*R^2
30 PRINT "THE AREA OF THE CIRCLE IS";A
40 PRINT
50 GOTO 10
WHAT IS THE RADIUS? 7.4
THE AREA OF THE CIRCLE IS 172.0336
```

Note that line 20 in the above example makes use of the built-in PRINT statement contained within INPUT.

INPUT# Statement

Purpose:

To read data items from a communications port and assign them to program variables.

Syntax:

INPUT# ***device number, variable list***

Comments:

device number is either 1 or 2 depending upon which communications port is to be addressed.

variable list contains the variable names to be assigned to the items in the device.

The data items returned by the device appear just as they would if data were being typed on the keyboard in response to an INPUT statement.

The variable type must match the type specified by the variable name.

With INPUT#, no question mark is printed, as it is with INPUT.

Numeric Values

For numeric values, leading spaces and line feeds are ignored. The first character encountered (not a space or line feed) is assumed to be the start of a number. The number terminates on a carriage return, line feed, or comma.

Strings

If BASIC is scanning the sequential data for a string, leading spaces and line feeds are ignored.

If the first character is a double quotation mark ("), the string will consist of all characters read between the first double quotation mark and the second. A quoted string may not contain a double quotation mark as a character. The second double quotation mark always terminates the string.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

If the first character of the string is not a double quotation mark, the string terminates on a comma, carriage return, or line feed, or after 128 characters have been read.

INPUT$ Statement

Purpose:

To return a string of *x* characters read from the auxiliary ports.

Syntax:

INPUT$ *#device number,n,variable$*

Comments:

device number is the communications port, COM1 or COM2. *n* specifies the number of characters to be read, which are read into *variable$* as a string. CTRL-C aborts the operation.

The INPUT$ function is preferred over INPUT and LINE INPUT statements for reading communications ports, because all ASCII characters may be significant in communications. INPUT is the least desirable because input stops when a comma or carriage return is seen. LINE INPUT is more desirable because it terminates when a carriage return is received.

Example:

```
.
.
.
40 INPUT$ #1,10,A$
50 PRINT A$
.
.
.
```

The above statements will read precisely 10 characters from COM1 and echo them to the screen.

INSTR Function

Purpose:

To search for the first occurrence of string y$ in x$, and return the position at which the string is found.

Syntax:

INSTR*(x$,y$ [,n])*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

Optional offset *n* sets the position for starting the search. The default value for *n* is 1.

If *n* equals zero or greater than 128, the error message **"out of range"** is returned.

INSTR return 0 if

n > LEN(x$)
x$ is null
y$ cannot be found

If y$ is null, INSTR returns *n*.

x$ and y$ may be string variables, string expressions, or string literals.

Example:

```
10 X$="ABCDEBXYZ"
20 Y$="B"
30 PRINT ,INSTR(X$,Y$), INSTR(X$,Y$,4)
RUN
      2            6
Ready
```

The interpreter searches the string "ABCDFBXYZ" and finds the first occurrence of the character B at position 2 in the string. It then starts another search at position 4 (D) and finds the second match at position 6 (B). The last three characters are ignored, since all conditions set out in line 30 were satisfied.

INT Function

Purpose:

To truncate an expression to a whole number.

Syntax:

INT*(x)*

Comments:

Negative numbers return the next lowest number.

The FIX function also returns an integer value.

Examples:

```
PRINT INT(98.89)
98
Ready

PRINT INT(-12.11)
-13
Ready
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

IO24 Statement

Purpose:

To specify the conditions on the 24-bit parallel I/O port which, when satisfied, will generate an interrupt to a specified program line number or label or screen name.

Syntax:

IO24 *v$*
IO24 OFF
IO24 STOP
IO24 CONT
(SEE ALSO CALL LOWBITS(*mode*))

Comments:

Depending upon the conditions determined by the CALL LOWBITS statement, the IO24 statement will generate an interrupt when group of bits on the I/O port match a specified pattern, go low, or change state. The default mode of operation is a pattern match. If any other mode is desired (low or change state modes), refer to the CALL LOWBITS statement.

PATTERN MODE:

v$ is a string expression that specifies a bit pattern on a 24-bit parallel I/O port which when satisfied, generates a trap to a specified program line number or label or screen name. 0, 1, and X are the only characters allowed in the string. X's within the string indicate that the specified bit can either be a 1 or a 0. If the string contains less than 24 digits, the rightmost character in the string is treated as the LSB on the 24-bit parallel I/O port. All unspecified bits in the leftmost portion of the string will be treated as X's.

If *v$* = "0011XX1100" a trap will occur when I/O bits 0, 1, 8 and 9 are low and I/O bits 2,3,6 and 7 are high. The state of I/O bits 4 and 5 do not matter.

LEVEL MODE:

v$ is specified identically as in the pattern mode. 0's in the string specify the bit positions which, when low will generate an interrupt. This is an "OR" condition. Only one of the bits specified by a 0 need be low to generate an interrupt.

If *v$* is "0110", and CALL LOWBITS(1) has been issued, an interrupt will occur if I/O bits 0 or 3 are low. Note that another interrupt will occur if I/O bits do not change back to 1 prior to re enabling the interrupts.

CHANGE MODE:

v$ is specified identically as in the pattern mode. 0's in the string specify the bit positions which, when changed from a 1 to a 0 or from a 0 to a 1 will generate an interrupt. This is an "exclusive OR" condition. Only the one of the bits specified by a 0 need change state to generate an interrupt.

If *v$* is "0110", and CALL LOWBITS(2) has been issued, an interrupt will occur if I/O bits 0 or 3 change from a 1 to a 0 or from a 0 to a 1.

IO24 OFF turns off I/O trapping.

IO24 STOP suspends the I/O trapping until the IO24 CONT statement is issued.

IO24 CONT reactivates the I/O trapping. If conditions which would satisfy an I/O trap are met, but an IO24 STOP has been issued, issuing an IO24 CONT will result an I/O trap.

NOTE:
IO24 traps are disabled (OFF) after an event trap has occurred, and must be re-started by issuing an IO24 string expression.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

```
10 ' set up output port for all 1's
20 out -1
30 call lowbits(2)
40 on io24 gosub 100
50 io24 "1001"
60' wait for interrupt
70 goto 70
90 ' I/O interrupt routine
100 print "io24 interrupt has occurred"
110 io24 "1001"
120 return
```

This example sets up the parallel I/O interrupt to occur if either bits 1 or 2 change state (0 to 1 or 1 to 0). Note that after an I/O interrupt has occurred, another IO24 *v$* statement must be issued to restart the system.

KEY Statement

Purpose:

To allow special tasks to be performed with one keystroke by interrupting the current running program. Upon completion, program resumes to the normal operation.

Syntax:

KEY ON
KEY OFF
KEY STOP
KEY CONT
KEY LIST
KEY LLIST

Comments:

Function keys can only be redefined in execution mode. (See KEY*(n)* in the next section.) In the editor they are assigned the following and cannot be changed:

F1-RUN F2-LIST F3-REMOT F4-CONT F5-AUTO F6-EDIT

F1, F2, and F4 execute the commands; F3 toggles the remote mode; F5 and F6 print the commands and allow the user to enter parameters.

KEY ON and KEY OFF (in editing mode)

Turns on or off the bottom line on the screen that displays the function keys. This allows the user to have an extra line on the display.

Any other use refers to redefining the usage of function keys in the program execution.

KEY ON (in running mode)

Displays the six strings that are defined previously by the user on the 8th or last line on the screen. If none of the function keys are defined, this statement will have no effect.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

KEY OFF (in running mode)

Erases the key display from the 8th line, making that line available for program use. KEY OFF does not disable the function keys.

KEY STOP

Suspends function key interrupts from all ten function keys. Function key interrupts are remembered, and when a KEY CONT is issued, they are processed.

KEY CONT

Resumes function key interrupts. KEY CONT is used after a KEY STOP to resume function key operation.

KEY LIST

Lists all 6 string expressions on the screen. All 5 characters are displayed.

KEY LLIST

Same as KEY LIST except output is directed to the printer.

Examples:

```
KEY OFF
```

Erases the bottom line on the display. Screen can now use 8 lines instead of 7 lines.

```
KEY ON
```

Turns back on the key display.

```
10 KEY 1, "ACCEL"
20 KEY 2, "VEL"
30 KEY 3, "DIST"
40 KEY 4, "EXIT"
50 KEY OFF
.
.
.
70 KEY ON
.
.
.
```

Function keys are defined in the above program on statement lines 10 through 40. The statement on line 50 erases them from the display. Line 70 turns the line back on.

NOTE:
KEY traps are automatically suspended (STOPPED) after they occur. It is the programmer's responsibility to re-enable KEY traps after the trap subroutine has been executed. The usual method for re-enabling the traps is to place a CONT statement just prior to the RETURN for the trap subroutine.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

KEY(n) Statement

Purpose:

To define, initiate, or terminate key capture in a BASIC program to perform a defined task.

Syntax:

KEY*(n)*, *string expression*
KEY*(n)*, ON
KEY*(n)*, OFF
KEY*(n)*, STOP
KEY*(n)*, CONT
KEY*(n)*, LIST
KEY*(n)*, LLIST

Comments:

n is a number from 1 to 10 that indicates which function key is to be captured. *String expression* is the message that describes briefly what the task is to be performed.

KEY*(n), string expression* defines the message to be displayed on the key line, but it is not required to use the function keys.

Execution of the KEY*(n)*, ON statement is required to activate keystroke capture from the function keys (function key interrupts). When the KEY*(n)*, ON statement is activated and an ON KEY*(n)* is defined, BASIC checks each new statement to see if the specified key is pressed. If so, BASIC performs a GOSUB to the line number or label or screen name specified in the ON KEY*(n)* statement.

Once KEY*(n)*, STOP is executed, function key interrupts are suspended. If a function key which is stopped is pressed, executing a KEY*(n)*, CONT will re-enable function key interrupts, and in this case generate a function key interrupt.

When KEY*(n)*, OFF is executed, no key capture occurs and no keystrokes are retained.

KEY*(n),* LIST and KEY*(n),* LLIST print the string corresponding to the function key*(s)* on the screen and printer respectively.

For further information on key trapping, see the ON KEY*(n)* statement.

NOTE:
KEY traps are automatically suspended (STOPPED) after they occur. It is the programmer's responsibility to re-enable KEY traps after the trap subroutine has been executed. The usual method for re-enabling the traps is to place a KEY (n), CONT statement just prior to the RETURN for the trap subroutine.

LABEL Statement

Purpose:

Allocate a label on the present line.

Syntax:

LABEL *labelname*

Comments:

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

labelname is an eight character alpha label which names the present line. *labelname* can be used in place of line numbers in GOTO or GOSUB statements.

Example:

```
10 GOSUB IOFUNC
20 GOSUB DSPFUNC
30 END
100 LABEL IOFUNC
110 OUT MAP "1010"
120 RETURN
200 LABEL DSPFUNC
210 CLS
220 PRINT "DISPLAY FUNCTION"
230 RETURN
RUN
```

Line 100 is called, and I/O bits 3 and 1 are set to a 1 in line 110. Line 200 is called, the display is cleared and a message is placed on the screen. Using labels allows program flexibility, since line numbers for subroutines can be replaced by meaningful names.

LEFT$ Function

Purpose:

To return a string that comprises the leftmost *n* characters of X$.

Syntax:

v$*=LEFT$*(x$, n)

Comments:

n must be within the range of 0 to 127. If *n* is greater than LEN(X$), the entire string (x$) will be returned. If *n* equals zero, the null string (length zero) is returned (see the MID$ and RIGHT$ substring functions).

Example:

```
10 A$="BASIC"
20 B$=LEFT$(A$,3)
30 PRINT B$
RUN
BAS
Ready
```

LEN Function

Purpose:

To return the number of characters in X$.

Syntax:

LEN*(x$)*

Comments:

Non printing characters and blanks are counted.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

x$ is any string expression.

```
10 X$="PORTLAND, OREGON"
20 PRINT LEN (X$)
16
Ready
```

Note that the comma and space are included in the character count of 16.

LINE INPUT Statement

Purpose:

To input an entire line (up to 78 characters) from the keyboard into a string variable, ignoring delimiters.

Syntax:

LINE INPUT *[prompt string;]string variable*

Comments:

prompt string is a string literal, displayed on the screen, that allows user input during program execution.

A question mark is not printed unless it is part of *prompt string*.

string variable accepts all input from the end of the prompt to the carriage return. Trailing blanks are ignored.

LINE INPUT is almost the same as the INPUT statement, except that it accepts special characters (such as commas) in operator input during program execution.

If a line-feed/carriage return sequence is encountered, both characters are input and echoed. Data input continues.

If LINE INPUT is immediately followed by a semicolon, pressing the RETURN key will not move the cursor to the next line.

A LINE INPUT may be escaped by typing CTRL-C. BASIC returns to command level and displays Ready.

Typing CONT resumes execution at the LINE INPUT line.

Example:

```
100 LINE INPUT A$
```

Program execution pauses at lines 100, and all keyboard characters typed thereafter are input to string A$ until RETURN, CTRL-M, or CTRL-C is entered.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

LINE INPUT # Statement

Purpose:

To input an entire line (up to 128 characters) from a communications port into a string variable, ignoring delimiters.

Syntax:

LINE INPUT #*device number,string variable*

Comments:

device number is the specified communications port 1 or 2.

Unlike LINE INPUT, (same name without a # after it), a prompt string or question mark is not printed.

string variable accepts all input from the end of the prompt to the carriage return. Trailing blanks are ignored.

LINE INPUT # is almost the same as the INPUT # statement, except that it accepts special characters (such as commas) from the communications device.

If a line-feed/carriage return sequence is encountered, the input into *string variable* is terminated, and the line-feed is placed into *string variable* as the last character. If a carriage return/line-feed sequence is encountered, the input is terminated when the carriage return is encountered, and the next LINE INPUT # or INPUT $ will end up with a carriage return as the first character in it. For this reason the programmer may wish to "strip" line-feeds from the communications device by using the CONFIG command.

A LINE INPUT # may be escaped by typing CTRL-C. BASIC returns to command level and displays Ready.

Typing CONT resumes execution at the LINE INPUT # line.

Example:

```
100 LINE INPUT #1,A$
```

Program execution will pause at line 100 until a string followed by a carriage return is encountered. The string will be returned (excluding the carriage return) in A$. A CNTL-C will terminate line 100 and return to the command level, displaying Ready.

LIST Command

Purpose:

To list all or part of a program to the screen.

Syntax:

LIST *[linenumber][-linenumber]*
LIST *[linenumber-]*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

linenumber is a valid line number within the range of 0 to 65535.

Use the hyphen to specify a line range. If the line range is omitted, the entire program is listed. *linenumber*- lists that line and all higher numbered lines. -*linenumber* lists lines from the beginning of the program through the specified line.

The period (.) can replace either *linenumber* to indicate the current line.

Any listing may be interrupted by pressing CTRL-C.

Examples:

```
LIST
```

Lists all lines in program.

```
LIST -20
```

Lists 1 through 20.

```
List 10-20
```

Lists 10 through 20

```
List 20-
```

Lists lines 20 through the end of the program.

LLIST Command

Purpose:

To list all or part of the program currently in memory to the line printer.

Syntax:

LLIST *[linenumber][-linenumber]*
LLIST *[linenumber-]*

Comments:

BASIC always returns to command level after a LLIST is executed. The line range options for LLIST are the same as for LIST.

Examples:

See the examples in the LIST statement.

LOC Function

Purpose:

To determine the number of bytes in the selected COM port's receive buffer

Syntax:

LOC (*port*)

Comments:

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

This command will let you know how many bytes are in the specified COM buffer NOTE:

port is the COM port buffer you wish to check.

Examples:

```
Print LOC(1)
7
Line input #1,a$
Print a$
hello
```

The first line gets the number of bytes in COM 1's receive buffer and prints it. The second print statement prints the actual string that is in COM 1. The 7 bytes found by the LOC command are the five characters in the word "hello", a carriage return, and a line feed.

LOCK Command

Purpose:

To lock a program into memory keeping it from being viewed, altered, or tampered with in any manner.

Syntax:

LOCK [(on)]

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

The LOCK command locks a program into memory in such a manner that it cannot be viewed, altered, or tampered with in any way. The screen will display the following message:

```
--------- WARNING ! ---------
THIS COMMAND WILL LOCK THE SYSTEM
SOFTWARE FROM FURTHER CHANGES.
PROCEED WITH LOCK? (YES/NO):
```

WARNING!
It is NOT possible to unlock a program once it has been locked. Before locking your program, it is HIGHLY advisable to upload it to a PC and store it on disk. To completely erase the contents of memory, thereby erasing the program, press CTRL-ALT-DEL keys simultaneously, or use the ApplicationBuilder's RESET feature to reset the 1000 Series product.

Lock (on) will lock the program in memory without displaying and asking the question.

Examples:

```
LOCK
--------- WARNING ! ---------
THIS COMMAND WILL LOCK THE SYSTEM
SOFTWARE FROM FURTHER CHANGES.
PROCEED WITH LOCK? (YES/NO): YES
Ready
LIST
system is locked
Ready
EDIT .
system is locked
Ready
LLIST
system is locked
Ready
NEW
system is locked
Ready
```

LOG Function

Purpose:

To return the natural logarithm of *x*.

Syntax:

LOG*(x)*

Comments:

x must be a number greater than zero.

Examples:

```
PRINT LOG(2)
.693147
Ready

PRINT LOG(1)
0
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
Ready
```

LPRINT and LPRINT USING Statements

Purpose:

To print data at the line printer.

Syntax:

LPRINT ***[list of expressions][;]***
LPRINT USING ***string exp; list of expressions(;)***

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

list of expressions consists of the strong or numeric expression separated by semicolons.

string expressions is a string literal or variable consisting of special formatting characters. The formatting characters determine the field and the formal of printed strings or numbers.

These statements are the same as PRINT and PRINT USING, except that output goes to the line printer. For more information about string and numeric fields and the variables used in them, see the PRINT and PRINT USING statements.

The LPRINT and LPRINT USING statements assume that your printer is an 80-character-wide printer.

MID$ Function

Purpose:

To return a string of *m* characters from x$, beginning with the *n*th character.

Syntax:

MID$*(x$, n [,m])*

Comments:

n must be within the range of 1 to 128.

m must be within the range of 0 to 127.

If *m* is omitted, or if there are fewer than *m* characters to the right of *n,* all rightmost characters beginning with *n* are returned.

If *n*>LEN(x$), the MID$ function returns a null string.

If *m* equals 0, the MID$ function returns a null string.

If either *n* or *m* is out of range, an **"out of range"** error is returned.

For more information and examples, see the LEFT$ and RIGHT$ functions.

Example:

```
10 A$="GOOD"
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,8,8)
RUN
GOOD EVENING
Ready
```

Line 30 concatenate (joins) the A$ string to another string with a length of eight characters, beginning at position 8 within the B$ string.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

MID$ Statement

Purpose:

To replace a portion of one string with another string.

Syntax:

MID$*(stringexp1,n[,m])=stringexp2*

Comments:

Both *n* and *m* are integer expressions.

stringexp1 and *stringexp2* are string expressions.

The characters in *stringexp1,* beginning at position *n,* are replaced by the characters in *stringexp2*.

The optional *m* refers to the number of characters from *stringexp2* that are used in the replacement. If *m* is omitted, all of *stringexp2* is used.

Whether *m* is omitted or included, the replacement of characters never goes beyond the original length of *stringexp1*.

Example:

```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
Ready
```

Line 20 overwrites "MO" in the A$ string with "KS".

NEW Command

Purpose:

To delete the program currently in execution memory and clear all variables.

Syntax:

NEW

Comments:

NEW is entered at command level to clear memory before entering a new program. BASIC always returns to command level after a NEW is executed.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Examples:

```
NEW
Ready
```

or

```
980 PRINT "DO YOU WISH TO QUIT (Y/N)
990 ANS$=INKEY$: IF ANS$=""THEN 990
1000 IF ANS$="Y" THEN NEW
1010 IF ANS$="N" THEN 980
1020 GOTO 990
```

OCT Function

Purpose:

To convert an octal string to decimal value.

Syntax:

OCT*(x$)*

Comments:

x$ is a string that contains the octal string. Any illegal characters such as a letter or the digit 9 will terminate evaluation at that point and the result will be the conversion up to that part of the string. This statement converts a decimal value within the range of -2147483647 to +2147483647 to an octal string expression. OCT$ is the complement of this function.

Example:

```
PRINT OCT( "1234" )
668
Ready
```

OCT$ Function

Purpose:

To convert a decimal value to an octal value.

Syntax:

OCT$*(x)*

Comments:

x is rounded to an integer before OCT$*(x)* is evaluated. This statement converts a decimal value within the range of -2147483647 to +2147483647 to an octal string expression. Octal numbers are numbers to base 8 rather than base 10 (decimal numbers). OCT is the complement of this function. See the BIN$ and HEX$ functions for binary and hexadecimal conversions.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

```
10 PRINT OCT$(18)
RUN
22
Ready
```

Decimal 18 equals octal 22.

ON COM(n), ON IO24, ON KEY(n), ON TIMER(n) Statements

Purpose:

To create an event trap that will gosub to a line number or label for a specified event (such as communications or pressing function keys).

Syntax:

ON *event specifier* GOSUB *line number or label*

Comments:

The syntax shown sets up an event trap line number or label for the specified event. A *line number* of 0 disables trapping for this event.

Once trap line numbers or labels have been set, event trapping itself can be controlled with the following syntax lines:

event specifier	**ON**	When an event is ON, and a non zero line number or label is specified for the trap, then every time BASIC starts a new statement, it checks to see if the specified event has occurred. If it has, BASIC performs a GOSUB to the line specified in the ON statement. ON does not apply to TIMER traps.
event specifier	**OFF**	This will clear the interrupt flags for each character received. When an event is OFF, no trapping occurs and the event is not remembered, even if it occurs.
event specifier	**STOP**	When an event is stopped, no trapping can occur, but if the event happens, it is remembered so an immediate trap occurs when an event specifier ON is executed.
event specifier	**CONT**	Resume event trapping, allowing events which occurred while STOPPED to occur.

NOTE:
TIMER and IO24 traps are disabled (OFF) after an event trap has occurred, and must be re-started by issuing a TIMER*(n), time* (see the TIMER statement) or IO24 *string expression* (see the IO24 statement).

NOTE:
COM and KEY traps are automatically suspended (STOPPED) after they occur. It is the programmer's responsibility to re-enable COM or KEY traps after the trap subroutine has been executed. The usual method for re-enabling the traps is to place a CONT statement just prior to the RETURN for the trap subroutine.

When an error trap takes place, all traps are automatically disabled.

Traps will never take place when BASIC is not executing a program.

The following are valid values for *event specifier:*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

COM*(n)* *n* is the number of the COM channel (1 or 2) (see COM*(n)* statement).

IO24 Trap on 24-bit I/O port bit pattern (see IO24 statement).

KEY*(n)* *n* is a function key number 1-10, the function keys F1 through F10 (see KEY*(n)* statement).

TIMER*(n)* *n* is a numeric expression which specifies which timer will generate the trap. *n* specifies the range of 1 to 8. A value outside of this range results in an **"illegal parameter"** error. See the TIMER command for a description of the timer setup parameters.

RETURN *line number or label* This optional form of RETURN is primarily intended for use with event trapping. The event-trapping routine may want to go back into the BASIC program at a fixed line number or label while still eliminating the GOSUB entry that the trap created.

Use of the non-local RETURN must be done with care. Any other GOSUB, WHILE, or FOR that was active at the time of the trap remains active.

If the trap comes out of a subroutine, any attempt to continue loops outside the subroutine results in a **"next without for"** error.

Special Notes About Each Type of Trap

COM Trapping

Typically, the COM trap routine will read an entire message from the communications port before returning.

It is NOT recommended that you use the COM trap for single character messages, since at high baud rates the overhead of trapping and reading for each individual character may allow the interrupt buffer for COM to overflow. Remember to re-enable traps after executing the trap subroutine. The usual method for re-enabling the traps is to place a COM*(n)*, OFF statement followed by the ON COM statement again just prior to the RETURN for the trap subroutine.

Key Trapping

Function keys 1 through 10 are trappable.

Refer to the KEY(*n)* statement for more information.

No type of trapping is activated when BASIC is in editing mode. Function keys resume their standard expansion meaning during input. Remember to re-enable traps after executing the trap subroutine. The usual method for re-enabling the traps is to place a KEY*(n)*, CONT statement just prior to the RETURN for the trap subroutine.

TIMER*(n)* Trapping

An ON TIMER*(n)* event trapping statement is used with applications needing an internal timer. The TIMER trap occurs when TIMER*(n)* has timed out. The 1000 SERIES PRODUCT has 8 timers. Timers 1 through 4 count milliseconds, while timers 5 through 8 count seconds. See the TIMER command for a description of how to set up these timers. The ON TIMER statement allows the program to have up to 8 timers active at any given time. Timers are disabled after a trap occurs. They must be re-enabled by issuing a TIMER *(n), time* statement.

IO24 Trapping

An ON IO24 event trapping statement is used to cause the program to execute a trap routine when the 24-bit parallel I/O satisfies a specific bit pattern. For more details see the IO24 statement. The IO24 trap is disabled after it occurs. It must be re-enabled by issuing a IO24 *string expression* statement.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example 1 - Communications Trapping:

This is a builder program that waits for the user to enter five characters from a terminal followed by a return character, then echoes the received string to the screen.

```
- GOTO SCREEN begin
*>SCREEN begin
- COM: INIT COM 2 ECHO 1 BAUD 96 HANDSHAKE 0 TIMEOUT 0 STRIP 1
- COM: WHEN COM 2 GETS A CHAR, GOSUB datain1
- HOT KEY (6) "STOP" INTERRUPT: GOSUB LABEL turnoff
- PUT TEXT AT (1,3): "# Characters"
- PUT TEXT AT (8,4): "data:"
- SOFTKEY (1) "stop" GOTO SCREEN turnoff
- SOFTKEY WAIT
  STOP
- LABEL datain1
- PUT TEXT AT (1,1): "IRQ"
- LABEL datain2
  a%=loc(2)
- PUT NUMBER a% AT (13,3) USING "###"
  if a%>4 then goto datain
  goto datain2
- LABEL datain
- COM: GET COM 2 TEXT datarec$
- PUT TEXT AT (13,4): datarec$
  DELAY 100:POS 1,6
  REPEAT:X=INKEY(#2):PRINT X;:UNTIL X=0
- PUT TEXT AT (1,1): "    "
  COM(2),OFF
- COM: WHEN COM 2 GETS A CHAR, GOSUB datain1
- RETURN
- LABEL turnoff
  STOP
- END OF PSEUDOCODE
```

Example 2 - 24-bit I/O trapping:

```
10 ' SET UP OUTPUT PORT FOR ALL 1'S
20 OUT -1
30 ON IO24 GOSUB 100
40 ' WAIT FOR INTERRUPT
50 IRQ=0
60 IO24 "010"
70 IF IRQ=0 THEN GOTO 70
80 GOTO 50
90 ' I/O INTERRUPT ROUTINE
100 PRINT "IO24 INTERRUPT HAS OCCURRED"
110 IRQ=1
120 RETURN
```

Example 3 - Softkey Trapping:

```
10 ON KEY (1) GOSUB 1000
20 ON KEY (2) GOSUB 2000
30 KEY (1), "FN-1"
40 KEY (2), "FN-2"
50 KEY (1), ON
60 KEY (2), ON
70 CLS
80 KEY STOP
90 POS 2,3
100 PRINT "WAITING FOR F-KEY"
110 KEY CONT
120 GOTO 80
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
1000 POS 1,5
1010 PRINT "F1 KEY IS PRESSED"
1020 KEY (1),CONT
1030 RETURN
2000 POS 1,5
2010 PRINT "F2 KEY IS PRESSED"
2020 KEY (2),CONT
2030 RETURN
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example 4 - Timer trapping:

```
10 CLS
20 DIM T%(8), N%(8)
30 FOR X% = 1 TO 8:READ T%(X%):NEXT
100 ON TIMER(1) GOSUB 1100
110 ON TIMER(2) GOSUB 1200
130 ON TIMER(3) GOSUB 1300
140 ON TIMER(4) GOSUB 1400
150 ON TIMER(5) GOSUB 1500
160 ON TIMER(6) GOSUB 1600
170 ON TIMER(7) GOSUB 1700
180 ON TIMER(8) GOSUB 1800
190 TIME$ = "12AM"
200 FOR X% = 1 TO 8
210 TIMER X%,T%(X%)
220 NEXT
800 IF INKEY$() = "" THEN 800
810 TIMER OFF
820 CLS:END
1100 TIMER STOP:QT%=1:GOTO 2000
1200 TIMER STOP:QT%=2:GOTO 2000
1300 TIMER STOP:QT%=3:GOTO 2000
1400 TIMER STOP:QT%=4:GOTO 2000
1500 TIMER STOP:QT%=5:GOTO 2000
1600 TIMER STOP:QT%=6:GOTO 2000
1700 TIMER STOP:QT%=7:GOTO 2000
1800 TIMER STOP:QT%=8:GOTO 2000
2000 POS 1,QT%
2010 N%(QT%) = N%(QT%)+1
2020 PRINT "T";QT%;": ";TIME$(2);N%(QT%);
2030 TIMER QT%,T%(QT%)
2040 TIMER CONT:RETURN
3000 DATA 100,1500,2200,3700,1,2,3,4
```

ON ERROR GOTO Statement

Purpose:

To enable error trapping and specify the first line of the error-handling subroutine.

Syntax:

ON ERROR GOTO *line number* or *label* or *screen name*

Comments:

Once error trapping has been enabled, all errors detected by BASIC, including direct mode errors (for example, syntax errors), cause BASIC to branch to the line in the program that begins the specified error-handling subroutine.

BASIC branches to the line specified by the ON ERROR statement until a RESUME statement is found.

If *line number* or *label* or *screen* name does not exist, an **"undefined line"** error results.

To disable error trapping, execute the following statement:

```
ON ERROR GOTO 0
```

Subsequent errors print and error message and halt execution.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

An ON ERROR GOTO 0 statement in an error-trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

If an error occurs during execution of an error-handling subroutine, the BASIC error message is printed and execution terminated. Error trapping does not occur within the error-handling subroutine.

Example 1: Simple error trapping

```
10 ON ERROR GOTO 1000
.
.
.
1000 A=ERR:B=ERL
1010 PRINT A,B
1020 RESUME NEXT
```

Line 1010 prints the type and location of the error on the screen (see the ERR and ERL variables).

Line 1020 causes program execution to continue with the line following the error.

Example 2: Intelligent error trapping with a PLC

The following is a section of code that will try and reestablish communication to a PLC in the event it is lost. It will try three times before giving up. If a different error occurred, it will print the error and line number, then exit the program

```
- GOTO SCREEN begin
*>SCREEN begin
  errcnt%=0
  maxerr%=3
  ON ERROR GOTO problem
  GOTO main

*>SCREEN main
- PUT LARGE TEXT AT (3,1): "THE MAIN SCREEN"
- PUT LARGE TEXT AT (1,2): "INIT-INITIALIZE PLC"
- SOFTKEY (6) "Exit" GOTO SCREEN exitscr
- SOFTKEY (2) "INIT" GOTO SCREEN plcinit
  call cursor(0)
- SOFTKEY WAIT

*>SCREEN plcinit
- PUT LARGE TEXT AT (1,1): "INITIALIZING PLC..."
  call cursor(0)
- PLC: INITIALIZE COMMUNICATIONS FOR PLC 1 OF TYPE 1
  GOTO main

*>SCREEN problem
  IF err<>53 THEN GOTO other
  errcnt%=errcnt%+1
  IF errcnt%>maxerr% THEN GOTO notalk
- PUT LARGE TEXT AT (8,1): "ERROR!"
- PUT LARGE TEXT AT (2,2): "CAN'T TALK TO PLC"
- PUT LARGE TEXT AT (2,3): "RETRYING... #"
- PUT LARGE NUMBER errcnt% AT (15,3) USING "#"
  call cursor(0)
- PLC: INITIALIZE COMMUNICATIONS FOR PLC 1 OF TYPE 1
  RESUME
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
*>SCREEN notalk
- SOFTKEY (6) "RETRY" GOTO SCREEN main
- PUT LARGE TEXT AT (2,3): "LOST COMMUNICATION"
- PUT LARGE TEXT AT (2,1): "CALL MAINTENANCE"
  call cursor(0)
  errcnt%=0
- SOFTKEY WAIT

  LABEL other
  PRINT "Error:";err;" on line ";erl
  LABEL exitscr
  CALL GCLS()
- END OF PSEUDOCODE
```

ON...GOSUB and ON...GOTO Statements

Purpose:

To branch to one of several specified line numbers or labels, depending on the value returned when an expression is evaluated.

Syntax:

ON *expression* GOTO *line numbers* or *labels* or *screen names*

ON *expression* GOSUB *line numbers* or *labels* or *screen names*

Comments:

In the ON...GOTO statement, the value of *expression* determines which line number or label or screen in the list will be used for branching. For example, if the value is 3, the third line number or label in the list will be the destination of the branch. If the value is a non integer, the fractional portion is rounded.

In the ON...GOSUB statement, each line number or label in the list must be the first line number of or label for a subroutine.

If the value of *expression* is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement.

Examples:

```
100 IF R<1 OR R>4 then print "ERROR":END
```

If the integer value of R is less than 1, or greater than 4, program execution ends.

```
200 ON R GOTO 150, 300, 320, 390
```

If R=1, the program goes to line 150.

If R-2, the program branches to line 300 and continues from there. If R=3, the branch will be to line 320, and so on.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

OUT Statement

Purpose:

To send a 24-bit data to the 24-bit I/O port.

Syntax:

OUT *variable&* **OUT XOR *variable&***
OUT AND *variable&* **OUT MAP *string***
OUT OR *variable&*

Comments:

variable& is a long integer variable. The most significant eight bits of the long integer will be ignored. The 24-bit value is output to the 24-bit I/O port.

OUT AND specifies that *variable&* will be logically AND'ed with the present contents of the 24-bit I/O port. This allows the program to CLEAR bits or fields of bits.

OUT OR specifies that *variable&* will be logically OR'ed with the present contents of the 24-bit I/O port. This allows the program to SET bits or fields of bits.

OUT XOR specifies that *variable&* will be exclusive OR'ed with the present contents of the 24-bit I/O port. This allows the program to TOGGLE bits or fields of bits.

OUT MAP allows a program to SET and CLEAR specific bits in on the 24-bit I/O port. *string* is a string variable which contains 1's, 0's, or X's. A 1 will set a bit, 0 will clear it, and an X will leave it unchanged. The contents of *string* generates a bitmap which sets or clears corresponding bits on the 24-bit parallel output port. For example:

```
OUT MAP "01XX"
```

clears bit 3 to a 0, sets bit 2 to a 1, and leaves bits 1 and 0 unchanged. Note that the right most part of the string always refers to the least significant bit of the 24-bit parallel output port. Any bits unspecified to the left of the bits specified in the string will be unchanged.

Examples:

```
100 OUT BIN$("10101010")
110 OUT AND BIN$("11110000")
120 OUT OR BIN$("0101")
```

This example was written using the BIN function so that the relationship between the bits can be seen. The first statement initializes the port to 10101010. The second statement clears the lower four bits and the port then contains 10100000. The third statement sets two bits, resulting in a 10100101 bit pattern on the port.

```
100 OUT MAP "1010XXXX"
```

In this example, the OUT MAP statement clears two bits (bits 4 and 6), and sets two bits (bits 5 and 7). Bits 0 through 3 and 8 through 23 remain unchanged.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

POS Statement

Purpose:

Move the cursor to the specified position.

Syntax:

POS *x,y*

Comments:

x is the x coordinate, or the column. *y* is the y coordinate, or the row. The range of *y* is from 1 to 8, 7 if the function key line is active. The range of *x* is 1 to 40.

Example:

```
10 CLS
20 POS 5,1
30 PRINT "THIS IS COLUMN 5"
```

Moves the cursor to the fifth column, first row and start printing.

POWER RESUME Command

Purpose:

On a power failure, specify whether to resume operation or stop.

Syntax:

POWER RESUME ON/OFF

Comments:

By issuing POWER RESUME ON, a program which is running when the power is removed will restart by itself when power is re-applied. If the 1000 SERIES PRODUCT was in the EDIT mode when the power was removed, the program will not resume operation, even with POWER RESUME ON.

By issuing POWER RESUME OFF, BASIC will return to the command mode after a power failure. The message **"power failed"** is sent to the screen when the power is restored.

PRINT Statement

Purpose:

To output a number on a string to the screen.

Syntax:

PRINT *[list of expressions][;]*
?[list of expressions][;]

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

If *list of expressions* is omitted, a blank line is displayed.

If *list of expressions* is included, the values of the expressions are displayed. Expressions in the list may be numeric and/or string expressions, separated by commas or semicolons. String constants in the list must be enclosed in double quotation marks.

For more information about strings, see the STRING$ function.

A question mark (?) may be used in place of the word PRINT when using the BASIC program editor.

Print Positions

BASIC divides the line into print zones of 8 spaces. The position of each item printed is determined by the punctuation used to separate the items in the list.

Separator	Print Position
,	Beginning of next zone
;	Immediately after last value

If a comma, semicolon, or TAB function ends an expression list, the next PRINT statement begins printing on the same line, accordingly spaced. If the expression list ends without a comma, semicolon, or TAB function, a carriage return is placed at the end of the lines (BASIC places the cursor at the beginning of the next line).

When numbers are printed on the screen, the numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus (-) sign. Single-precision numbers are represented with seven or fewer digits in a fixed-point or integer format.

See the LPRINT and LPRINT USING statements for information on sending data to be printed on a printer.

Example:

```
10 X$= STRING$("-",10)
20 PRINT X$;"MONTHLY REPORT";X$
----------MONTHLY REPORT------------
Ready
```

PRINT USING Statement

Purpose:

To print strings or numbers using a specified format.

Syntax:

PRINT USING *string expressions;list of expressions[;]*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

string expressions is a string literal or variable consisting of special formatting characters. The formatting characters determine the field and the format of printed strings or numbers.

list of expressions consists of the string or numeric expression separated by semicolons.

Numeric Fields

The following special characters may be used to format the numeric field:

A pound sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces) in the field.

. A decimal point may be inserted at any position in the field. If the format string specified that a digit is to precede the decimal point, the digit always is printed (as 0 if necessary). Numbers are rounded as necessary. For example:

```
PRINT USING "##.##";.78
0.78
READY

PRINT USING "###.##";987.654
987.65
READY

PRINT USING "##.##"  ;10.2,5.3,66.789,.234
10.20  5.30  66.79  0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ A plus sign at the beginning or end of the format string causes the sign of the number (plus or minus) to always be printed before or after the number.

- A minus sign at the beginning or end of the format string causes a minus sign to be printed before or after a negative number. A space is printed before or after a positive number.

0 A 0 at the beginning of the format string causes leading spaces in the numeric field to be filled with zeroes. A 0 is both a flag and/or a digit specifier. For example:

```
PRINT USING "0##.#";12.39, 0.9, 765.1
012.4 000.9 765.1
READY
```

$ A dollar sign at the beginning of the format string causes a dollar sign to be printed to the immediate left of the formatted string. For example:

```
PRINT USING  "$###.##";456.78
$456.78
READY
```

, A comma to the left of the decimal point in the format string causes a comma to be printed to the left of every third digit to the left of the decimal point.

```
PRINT USING "####.##";1234.5
1234.50
Ready
```

E E may be placed before the digit position characters to specify exponential format. Any decimal point position may be specified. The significant digits are left-justified, and the exponents are adjusted. Unless a leading + or trailing + or - is specified, one digit position is used to the left of the decimal point to print a space or a minus sign. For example:

```
PRINT USING "E##.##";234.56
23.46E+01
Ready

PRINT USING "E.####";888888
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
.889E6
Ready

PRINT USING "+E.##";123
+.12E3
Ready
```

Note that in the above examples, the comma is *not* used as a delimiter within the exponential format.

NOTE:
Overflow conditions, those conditions when the number being printed is either larger or smaller than the format string allows, convert to exponential numbers. For Example

```
PRINT USING "##.##";100
10.00E+1
Ready
```

In other words, make sure that your format string can accommodate the largest valid number your application will permit.

PRINT # and PRINT # USING Statements

Purpose:

To write data to a sequential device.

Syntax:

PRINT # ***device number*****,[USING*****string expressions;*****]*****list of expressions***

Comments:

device number is the number of the sequential communications port.

string expressions consists of the formatting characters described in the PRINT USING statement.

list of expressions consists of the numeric and/or string expressions to be written to the communications port.

Double quotation marks are used as delimiters for numeric and/or string expressions. The first double quotation mark opens the line for input; the second double quotation mark closes it.

If numeric or string expressions are to be printed as they are input, they must be surrounded by double quotation marks. If the double quotation marks are omitted, the value assigned to the numeric or string expression is printed. If no value has been assigned, 0 is assumed. The double quotation marks do not appear on the screen. For example:

```
10 PRINT#1,A
0

10 A=26
20 PRINT#1,A
26

10 A=26
20 PRINT#1,"A"
A
```

If double quotation marks are required within a string, use CHR$(34) (the ASCII character for double quotation marks). For example:

```
100 PRINT #1, "He said, "Hello", I think"
He said, 0, I think
```

Hello is replaced with a 0 because the machine assigns the value 0 the variable "Hello".

```
100 PRINT#1, "He said, "CHR$(34)"Hello, "CHR$(34) " I think."
He said, "Hello," I think
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

If the strings contain commas, semicolons, or significant leading blanks, surround them with double quotation marks. The following example will input "CAMERA" to A$, and "AUTOMATIC 93604-1" TO B$:

```
10 A$="CAMERA,AUTOMATIC":B$="93604-1"
20 PRINT#1,A$;B$
30 INPUT#1,A$,B$
```

To separate these strings properly, write successive double quotation marks using CHR$(34). For example:

```
40 PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$; CHR$(34)
"CAMERA,AUTOMATIC""93604-1"
```

The PRINT# does not compress data written to a communications port. An image of the data is written to the communications port, just as it would be displayed on the terminal screen with a PRINT statement. For this reason, be sure to delimit the data written to the communications port so that it may be read back correctly (i.e. terminate strings with delimiters).

In *list of expressions*, numeric expressions must be delimited by semicolons. For example.

```
PRINT#1,A;B;C;X;Y;Z
```

If commas are used as delimiters, the extra blanks inserted between print fields will also be written to the communications port. Commas have no effect, however, if used with the exponential format.

Strong expressions must be separated by semicolons in the list. To format the string expressions correctly on the communications port, use explicit delimiters in *list of expressions*. For example, the following:

```
10 A$="CAMERA":B$="93604-1"
20 PRINT#1,A$,B$
```

Outputs the following to communications port 1:

```
CAMERA93604-1
```

Because there are no delimiters, the receiving device may not recognize the transmission as two strings. If the receiving device uses a comma (,) for a delimiter, insert explicit delimiters into the PRINT# statement as follows:

```
30 PRINT#1,A$;",";B$
:
CAMERA,93604-1
```

PRN Statement

Purpose:

To turn on or off, or redirect the printer port.

Syntax:

PRN ON/OFF
PRN TO COM(*n*)

Comments:

PRN OFF disables the printer port (MODEL 1100 only)

PRN ON enables the printer port (MODEL 1100 only).

PRN TO COM(n), redirects the print output (LPRINT statements, TRACE PRN, etc.) to the COM port specified by *n*. n can be either 1 or 2.

Examples:

```
PRN ON
```
Enable printer

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

`PRN OFF`	Disable printer
`PRN TO COM(1)`	Printer output to COM port 1
`PRN TO COM(2)`	Printer output to COM port 2
`COM(1) TO PRN`	COM 1 outputs to printer
`COM(2) TO PRN`	COM 2 outputs to printer

READ Statement

Purpose:

To read value from a DATA statement and assign them to variables.

Syntax:

READ *list of variables*

Comments:

A READ statement must always be used with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a **"bad data"** results. A single READ statement may access one or more DATA statements. They are accessed in order. Several READ statements may access the same DATA statement. If the number of variables in *list of variables* exceeds the number of elements in the DATA Statement(s), an **"out of data"** message is printed.

If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ Statements begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored. To reread DATA statements from the start, use the RESTORE statements.

Example 1:

```
.
.
80 FOR I=1 TO 10
90 READ A (I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
..
.
.
```

This program segment reads the values from the DATA statements into array A. After execution, the value of A(1) is 3.08, and so on. The DATA statement (lines 110-120) may be placed anywhere in the program; they may even be placed ahead of the READ statement.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example 2:

```
10 PRINT "CITY";TAB(10);"STATE";TAB(20);"ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,","COLORADO",80211
40 PRINT C$;TAB(10);S$;TAB(20);Z
RUN
CITY  STATE ZIP
DENVER,     COLORADO     80211
Ready
```

This program reads string and numeric data from the DATA statement in line 30.

REM Statement

Purpose:

To allow explanatory remarks to be inserted in a program.

Syntax:

REM*[comment]*

Comments:

REM statements are not executed, but are output exactly as entered when the program is listed.

REM statements must stand alone on their own line. Do not append them onto the end of a command or statement.

Once a REM is encountered, the program ignores everything else until the next line number or program end is encountered.

REM statements may be branched into from a GOTO or GOSUB statement, and execution continues with the first executable statement after the REM statement. However, the program runs faster if the branch is made to the first statement.

Note:
Do Not Use REM In A Data Statement Because It Will Be Considered To Be Legal Data.

Examples:

```
.
.
120 REM CALCULATE AVERAGE VELOCITY
130 FOR 1=1 TO 20
440 SUM=SUM+V(I)
450 NEXT I
.
.
```

REMOTE Command

Purpose:

Connect or disconnect to a remote terminal or host computer.

Syntax:

REMOTE ON/OFF

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

REMOTE ON enables BASIC to receive input from the remote terminal or host computer through the communications port. The local keypad and/or keyboard will still be active. **REMOTE OFF** restores control to the remote terminal.

REMOTE ON/OFF can be issued from program statements. This is particularly useful to allow the programmer to issue a **REMOTE OFF** statement in a program to insure that the host computer or terminal is not receiving information that was just meant for the display.

Examples:

```
REMOTE ON
```

This releases control to the remote terminal.

```
REMOTE OFF
```

This regains control from the remote terminal.

RENUM Command

Purpose:

To renumber program lines.

Syntax:

RENUM[*new number*],[*old number*][,*increment*]]

Comments:

new number is the first line number to be used in the new sequence. The default is 10.

old number is the line in the current program where renumbering is to begin. The default is the first line of the program. *old number* cannot be greater than *new number*, otherwise an "**illegal arguments**" error will occur. To start on the first line, just leave out *old number*.

increment is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following ELSE, GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, RESTORE, RESUME, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message **"undefined line *x* in *y*"** appears. The incorrect line number reference *x* is not changed by RENUM, but line number *y* may be changed.

RENUM cannot be used to change the order of the program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65530. A **"line number overflow"** error results.

Examples:

```
RENUM
```

Renumbers the entire program. The first new line will be 10. Lines increment by 10.

```
RENUM 300,,50
```

Renumbers the entire program. The first new line number will be 300. Lines increment by 50.

```
RENUM 1000,900,20
```

Renumbers the lines from 900 up so they start with line number 1000 and are incremented by 20.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

REPEAT...UNTIL Statement

Purpose:

Execute a series of statements in a loop at LEAST once.

Syntax:

REPEAT

.

.

[loop statement]

.

.

UNTIL *expression*

Comments:

The loop statements are executed at least once. If *expression* is non zero (true), loop statements repeat execution until the *expression* evaluates to zero (false).

REPEAT .. UNTIL loops may be nested to any level. Each UNTIL matches the most recent REPEAT.

An unmatched UNTIL statement causes an **"until without repeat"** error. An unmatched REPEAT statement causes a **"repeat without until"** error.

A Warning About Breaking Out of REPEAT-UNTIL Loops:

REPEAT-UNTIL loops use the BASIC stack to "remember" where the top of the loop is. The REPEAT statement saves its line number on the stack so the UNTIL statement can figure out where to branch to if the ending conditions have not been met. For this reason, branching out of the REPEAT-UNTIL loop's body will leave the REPEAT statement's line number on the stack. This can create problems if this process is done repeatedly. Even worse, it can create a problem which normal testing would not catch, and only after months of operation will the program fail.

The indication of failure is the OUT OF MEMORY error. To avoid this failure, do not break out of a REPEAT-UNTIL loop. If the program must break out (or an error, or some other non-normal event), use a CALL CLRSTACK statement in the main body of the program to keep the stack from filling up.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

```
10 REPEAT
20 K% = INKEY()
30 UNTIL K% = 13
40 PRINT "FOUND ENTER KEY"
RUN
```

(user presses ENTER)

```
FOUND ENTER KEY
Ready
```

RESTORE Statement

Purpose:

To allow DATA statements to be reread from a specified line.

Syntax:

RESTORE*[line number]*

Comments:

If *line number* is specified, the next READ statement accesses the first item in the specified DATA statement.

If *line number* is omitted, the next READ statement accesses the first item in the first DATA statement.

Example:

```
10 READ A,B,C,
20 RESTORE
30 READ D,E,F
40 DATA 57,68,79
.
.
.
```

Assigns the value 57 to both A and D variables, 68 to B and E, and so on.

RESUME Statement

Purpose:

To continue program execution after an error-recovery procedure has been performed.

Syntax:

RESUME
RESUME NEXT
RESUME *line number* or *label* or *screen name*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

Any one of the four formats shown above may be used, depending upon where execution is to resume:

Syntax	Result
RESUME	Execution resumes at the statement that caused an error.
RESUME NEXT	Execution resumes at the statement immediately following the one that caused an error.
RESUME *line number or label*	Execution resumes at the specified line number.

A RESUME statement that is not in an error trapping routine causes a "**RESUME without error**" message to be printed.

Example:

```
110 ON ERROR GOTO 900
.
.
.
900 IF (ERR=230) AND (ERL=90)  THEN PRINT "TRY AGAIN":RESUME 80
.
.
.
```

If an error occurs after line 10 is executed, the action indicated in line 900 is taken and the program continues at line 80.

RETURN Statement

Purpose:

To return from a subroutine.

Syntax:

RETURN *[line number* or *label* or *screen name]*

Comments:

The RETURN statement causes BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement to return from different points in the subroutine. Subroutines may appear anywhere in the program.

RETURN *line number or label or screen name* is primarily intended for use with event trapping. It sends the event-trapping routine back into the BASIC program at a fixed line number or label or screen name while still eliminating the GOSUB entry that the trap created.

The nonlocal RETURN must be used with care. Any GOSUB, WHILE, or FOR statement active at the time of the trap remains active. Refer to the precautions associated with these statements regarding breaking out of a loop or failing to RETURN from a GOSUB.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

RIGHT$ Function

Purpose:

To return the rightmost *i* characters of string x$.

Syntax:

RIGHT$(*x$,i*)

Comments:

If *i* is equal to or greater than LEN(x$), RIGHT$ returns x$. If *i* equals zero, the null string (length zero) is returned (see the MID$ and LEFT$ functions).

Example:

```
10 B$="EASON BASIC"
20 PRINT RIGHT$(B$,5)
RUN
BASIC
Ready
```

Prints the rightmost five characters in the A$ string.

RND Function

Purpose:

To return a random integer between 0 and 32767.

Syntax:

RND*[(x)]*

Comments:

RND without argument generates a number between 0 and 32767. To get a random number within the range of zero through *n*, call RND(*n*). *n* must be from -32768 to 32767. If *n* = 0, a fractional number between 0 and 1 will be returned.

Example:

```
10 FOR I=1 TO 5
20 PRINT RND(101)
30 NEXT
RUN
53          30          31          51          5
Ready
```

Generates five pseudo-random numbers within the range of 0-100.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

RS422 Command

Purpose:

To enable or disable RS422 communications on a serial port.

Syntax:

RS422 #*n*, ON / OFF / TXON / TXOFF / TXAUTO

Comments:

n indicates the communications port: 1 is COM1, 2 is COM2.

RS422 #*n*, ON allows the RS422 receiver to accept data and enables the RS422 transmitter for port *n*.

RS422 #*n*, OFF disables transmit and receive and returns to RS232 operation.

RS422 #*n*, TXON enables the RS422 receiver and transmitter (same operation as RS422 #*n*, ON).

RS422 #*n*, TXOFF disables the RS422 transmitter (sets its outputs to a high-impedance state), and enables the RS422 receiver! Only in RS422 mode.

RS422 #*n*, TXAUTO is used to disable the RS422 transmitter except when a RS422 transmission is in operation. Specifying this mode, and printing to the communications port will automatically turn on the transmitter, transmit the data, wait two character times and turn off the transmitter. This state is useful for implementing an RS485 configuration, or to operate the unit in a multi-drop application.

RUN Command

Purpose:

To execute the program currently in memory.

Syntax:

RUN *[line number or label or screen name]*

Comments:

RUN or RUN *line number or label* runs the program currently in memory.

If *line number or label or screen name* is specified, execution begins on that line. Otherwise, execution begins at the lowest line number.

If there is no program in memory when RUN is executed, BASIC returns to command level.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

SGN Function

Purpose:

To return the sign of *x*.

Syntax:

SGN*(x)*

Comments:

x is any numeric expression.

If *x* is positive, SGN(x) returns 1.

If *x* is 0,SGN(x) returns 0

If *x* is negative, SGN(x) returns -1.

Example:

```
10 INPUT "Enter value",X
20 ON SGN(X)+2 GOTO 100,200,300
```

BASIC branches to 100 if **X** is negative, 200 if **X** is 0, and 300 if **X** is positive.

SIN Function

Purpose:

To calculate the trigonometric sine of *x*, in radians.

Syntax:

SIN*(x)*

Comments:

To obtain SIN*(x)* when *x* is in degrees, use SIN(*x**π/180).

Example:

```
PRINT SIN(1.5)
.9974948
Ready
```

The sine of 1.5 radians is .9974948 (single-precision)

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

SPACE$ Function

Purpose:

To return a string of *x* spaces.

Syntax:

SPACE$*(x)*

Comments:

x is rounded to an integer and must be within the range of 0 to 255.

Example:

```
10 FOR N=1 TO 5
20 X$=SPACE$(N)
30 PRINT X$;N
40 NEXT N
RUN
1
 2
  3
   4
    5
Ready
```

Line 20 adds one space for each loop execution.

SQR Function

Purpose:

Returns the square root of *x*.

Syntax:

SQR*(x)*

Comments:

x must be greater than or equal to 0.

SQR(*x*) is computed in single-precision unless the DOUBLE command is executed.

Examples:

```
10 FOR X=10 TO 25 STEP 5
20 PRINT X; SQR(X)
30 NEXT
RUN
10          3.162277
15          3.872982
20          4.472135
25          5
Ready
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

STOP Statement

Purpose:

To terminate program execution and return to command level.

Syntax:

STOP

Comments:

STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

Break in line *n*

BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command.

Example:

```
10 INPUT A,B,C
20 K=A^2*5.3;L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ready
PRINT L
30.76923
Ready
CONT
115.9
Ready
```

STR$ Function

Purpose:

To return a string representation of the value of *x*.

Syntax:

STR$*(x)*

Comments:

STR$*(x)* is the complementary function to VAL*(x$)* (see the VAL function).

Example:

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,40,50
.
.
```

This program branches to various subroutines, depending on the number of characters typed before the RETURN key is pressed.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

STRING$ Function

Purpose:

To return a string of multiple copies of the input string.

Syntax:

STRING$*(x$,n)*

Comments:

x$ is the string to be duplicated. *n* is the number of times *x$* gets duplicated.

Example:

```
10 X$ = STRING$ ( "ABC", 5 )
20 PRINT X$
RUN
ABCABCABCABCABC
Ready
```

TAB Function

Purpose:

Spaces to position *n* on the screen.

Syntax:

TAB*(n)*

Comments:

If the current print position is already beyond space *n,* TAB goes to that position on the next line.

Space 1 is the leftmost position. The rightmost position is the screen width. *n* must be within this range.

If the TAB function is at the end of a list of data items, BASIC will not return the cursor to the next line. It is as though the TAB function has an implied semicolon after it.

TAB may be used only in PRINT, LPRINT, or PRINT# statements.

Example:

```
10 PRINT "NAME"; TAB(25); "AMOUNT": PRINT
20 READ A$,B$
30 PRINT A$; TAB(25); B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                    AMOUNT
G. T. JONES             $25.00
Ready
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

TAN Function

Purpose:

To calculate the trigonometric tangent of *x,* in radians.

Syntax:

TAN*(x)*

Comments:

If TAN overflows, the **"overflow"** error message is displayed; machine infinity with the appropriate sign is supplied as the result, and execution continues.

To obtain TAN*(x)* when *x* is in degrees, use TAN(x*_/180).

Example:

```
PRINT TAN(PI/4)
 0.4142135
Ready
```

TIME Variable

Purpose:

To return the internal timer count.

Syntax:

v = TIME

Comments:

TIME returns the time since system reset to the nearest second based on a 24 hour clock. Therefore, the variable TIME rolls over to zero after 86400 seconds (24 hours x 60 minutes x 60 seconds). This is a read-only variable, and cannot be changed.

Note: disconnecting power from the 1000 Series device does not reset the time variable.

TIME is a reserved word and cannot be used as a variable in BASIC programs.

TIME$ Statement

Purpose:

To set or retrieve the current time.

Syntax:

TIME$ = *string exp*

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

string exp is a valid string literal or variable that lets you set hours *(hh),* hours and minutes *(hh:mm),* or hours, minutes, and seconds *(hh:mm:ss)* or optionally *(hh:mm:ss:am/pm).*

There are two forms for *string exp*:

24 hour format:

hh sets the hour (0-23). Minutes and seconds default to 00.
hh:mm sets the hour and minutes (0-59). Seconds default to 00.
hh:mm:ss sets the hour, minutes, and seconds (0-59).

12 hour format:

hh sets the hour (0-12). Minutes and seconds default to 00.
hh:mm sets the hour and minutes (0-59). Seconds default to 00.
hh:mm:ss sets the hour, minutes, and seconds (0-59).
hh:mm:ss:am/pm sets the hour, minutes, seconds and am/pm.

If *string exp* is not a valid string, an **"illegal arguments"** error result.

As you enter any of the above values, you may omit the leading zero, if any. You must, however, enter at least one digit. If you wanted to set the time as a half hour after midnight, you could enter TIME$= "0:30", but not TIME$=":30".

If any of the values are out of range, an **"illegal arguments"** error is issued. The previous time is retained.

Examples:

The following example sets the time at 8:00 A.M.:

```
TIME$ = "08:00"
Ready
PRINT TIME$(1)
08:00:05
Ready

TIME$ = "08:00:00:PM"
Ready
PRINT TIME$(2)
08:00:05 pm
```

TIME$ Function

Purpose:

To return the time in 12 or 24 hour format.

Syntax:

***string exp* = TIME$(*n*)**

Comments:

If ***n*** = 1, *string exp* will be "hh:mm:ss" in a 24 hour format.

If ***n*** = 2, *string exp* will be "hh:mm:ss am/pm" in a 12 hour format.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

```
PRINT TIME$(1)
14:23:04
Ready
PRINT TIME$(2)
2:23:05pm
```

TIMER Statement

Purpose:

To set the number of seconds or milliseconds that must occur before a timer interrupt is to occur.

Syntax:

TIMER(n), s
TIMER[(n)], OFF
TIMER[(n)], STOP
TIMER[(n)], CONT

Comments:

n specifies the timer number. Timers 1 through 4 count in milliseconds, while timers 5 through 8 count in seconds. Note that *n* is optional. By issuing a TIMER OFF, STOP, or CONT, all timers will be affected. This allows simple enabling and disabling of timer traps.

s specifies the number of seconds or milliseconds that must occur before a timer interrupt is to occur. For timers 1 through 4, *s* is in milliseconds, while for timers 5 through 8, *s* is in seconds.

TIMER*(n) s* turns a specified timer on for the specified duration allowing a timer trap to occur when it times out (reaches terminal count). An ON TIMER*(n)* statement must be issued to give direction to the timer trap, i.e. where to go when it times out. Note that when a timer times out, it is automatically turned off (same as TIMER*(n)*, OFF). It is the programmer's responsibility to restart the timer if another timer interrupt is desired.

OFF turns a specified timer off.

STOP suspends a timer trap. When a TIMER*(n)*, STOP is received, the timer keeps running. If it times out while stopped, the interrupt will be remembered. Issuing a TIMER*(n),* CONT will resume operation.

NOTE:
TIMER traps are disabled (OFF) after an event trap has occurred, and must be re-started by issuing a TIMER*(n), time* statement.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

```
10'Initialize timer parameters
20 on timer(1) gosub dotimer1
30 on timer(5) gosub dotimer5
40 timer 1,2000:'2000 milliseconds = 2 seconds
50 timer 5,4:'4 seconds
60'Endless loop until a key stroke is detected
70 print "Waiting for Timers..."
80 if inkey$() = "" then 80
90'Cleanup
100 timer off
110 end
120 label dotimer1
130 timer stop:'Prevent other timers to come in
140 print "Timer 1 interrupted at ";
150 print time$(2)
160'Timer 1 is reinitialized since it's turned off once triggered
170 timer 1,2000
180 timer cont:'Continue the other timers
190 return
200 label dotimer5
210'This is basically the same as Timer 1 except when reinitialized,
the
220'  count is in seconds instead.
230 timer stop
240 print "Timer 5 interrupted at ";
250 print time$(2)
260 timer 5,4:'4 seconds
270 timer cont
280 return
```

TRACE Command

Purpose:

To trace the execution of program statements.

Syntax:

TRACE *ON/OFF/PRN/COM(n)*

Comments:

ON turns the trace function on, while *OFF* turns the trace function off.

PRN and *COM(n)* imply that the trace function be turned on.

PRN redirects the trace output to the printer port, while *COM(n)* redirects it to the *COM* port specified by *n*.

As an aid in debugging, the TRACE ON/PRN/COM(n) command enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets.

TRACE ON/PRN/COM(n) may be executed in either the direct or indirect mode.

The trace flag is disabled with the *TRACE OFF* command, or when a NEW command is executed.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Example:

```
TRACE ON
Ready
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
RUN
[10] [20] [30] [40] 1 10 20
[50] [60] [30] [40] 2 20 30
[50] [60] [70]
Ready
TRACE OFF
Ready
```

VAL Function

Purpose:

Returns the numerical value of string x$.

Syntax:

VAL(*x$*)

Comments:

The VAL function also strips leading blanks, tabs, and line feeds from the argument string. For example, the following line returns -3:

The STR$ function (for numeric to string conversion) is the complement to the VAL(*x$*) function.

If the first character of *x$* is not numeric, the VAL(*x$*) function will return zero.

Example:

```
10 DATA "123","345","567"
20 FOR N=1 TO 3
30 READ A$
40 IF VAL(A$)=345 THEN PRINT N
50 NEXT
RUN
2
Ready
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

VER Function

Purpose:

Reports the software part number and version for the 1000 SERIES PRODUCT.

Syntax:

VER

Comments:

Reports the software version and part number on the 1000 SERIES PRODUCT display. The response to the VER command is in the following format

`20-0000X-YY-Z.ZZ`

Where *X* represents the product:

X	Product
3	Model 1100
2	Model 1000

Where *YY* represents the software option:

YY	Software Option
1	Standard Software
2	-MOD
3	-GE9
4	-TI3
5	-TI5
6	-PL5
7	-OM1

Where *Z.ZZ* represents the software version:

Example:

```
VER
20-00003-01-3.30
Ready
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

WHILE - WEND Statement

Purpose:

To execute a series of statements in a loop as long as a given condition is true.

Syntax:

WHILE expression

.

.

.

[loop statements]

.

.

.

WEND

Comments:

If *expression* is non zero (true), loop statements are executed until the WEND statement is encountered, BASIC then returns to the WHILE statement and checks *expression.* If it is still true, the process is repeated.

If it is not true, execution resumes with the statement following the WEND statement.

WHILE and WEND loops may be nested to any level. Each WEND matches the most recent WHILE.

An unmatched WHILE statement causes a **"while without wend"** error. An unmatched WEND statement causes a **"wend without while"** error.

A Warning About Breaking Out of WHILE-WEND Loops:

WHILE-WEND loops use the BASIC stack to "remember" where the top of the loop is. The WHILE statement saves its line number on the stack so the WEND statement can figure out where to branch to if the ending conditions have not been met. For this reason, branching out of the WHILE-WEND loop's body will leave the WHILE statement's line number on the stack. This can create problems if this process is done repeatedly. Even worse, it can create a problem which normal testing would not catch, and only after months of operation will the program fail.

The indication of failure is the OUT OF MEMORY error. To avoid this failure, do not break out of a WHILE-WEND loop. If the program must break out (or an error, or some other non-normal event), use a CALL CLRSTACK statement in the main body of the program to keep the stack from filling up.

Example:

```
10 WHILE K%<>13
20 K% = INKEY()
30 WEND
40 PRINT "FOUND ENTER KEY"
RUN
```

(user presses ENTER)

```
FOUND ENTER KEY
Ready
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CHAPTER 5. MODEL 1100 COUNTER INTERFACE

MODEL 1100 ONLY:

The Eason Technology MODEL 1100 counter option (options C01 and C02) allows the MODEL 1100 to count quadrature counter inputs, up/down pulses, or step and direction signals up to 1.0 MHz. The C01 and C02 provide a means for recording the position of an encoding device which generates A-B quadrature outputs. The C01 option is a single counter device, while the C02 provides two counters. Up to 32 bits of position information can be gathered and used by BASIC program statements. The power of the C02 option is supported by built-in basic CALL functions in the MODEL 1100. Refer to the MODEL 1100 MANUAL for more details on the hardware and specifications of the CO1 and C02.

CALL READCNT Command

MODEL 1100 ONLY:

Purpose:

Read contents of specified counter's position register.

Syntax:

CALL READCNT (*n,a&, mstimer&*)

Comments:

n is the desired counter to be read (1, 2, 3, or 4)

a& is the long integer variable that the count is to be placed into.

mstimer& is an optional parameter for reading the free running millisecond timer (useful for RPM & frequency calculations)

This command is only available on the Model 1100 with the C01 or C02 option(s) installed.

Example:

An encoder inputs 25 quadrature counts into channel 1 of a C02 Counter Interface. This will generate 100 counts in the counter since each pulse will generate 4 counts.

```
CALL READCNT (1,x&)
Ready
Print x&
100
Ready
```

CALL WRITECNT Command

MODEL 1100 ONLY:

Purpose:

To send commands to the Counter Interface.

Syntax:

CALL WRITECNT (*n, val&, cmd*)

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Comments:

n is the desired counter to be commanded (1 or 2).

val& is a long integer value which is used by some of the commands.

This command is only available on the Model 1100 with the C01 or C02 option(s) installed.

cmd is a counter instruction command:

0	Counter complete reset (both channels in C02)
1	Clear counter *n*
2	Select quadrature mode (default)
3	Select step and direction mode. A becomes the clock input, and B specifies the direction. A logic low (B+ high B- low) allows the counter to count up, while a logic high, (B+ low and B- high) allows the counter to count down. Counts occur when A goes from low to high.
4	UP / DOWN count mode. Counts up when A goes from low to high, down when B goes from low to high. B must be high to count up, A must be high to count down.
5	Z marker input resets counter when low (Z+ low, Z- high).
6	Z marker input gates counter input when low.
7	Unused.
8	Unused.
9	Disables Z marker (default)
10	Load counter offset. The offset is given by *val&* and loaded into counter *n*.
11	Select digital filter frequency. val& = 0 - 5 MHz (default - max bandwidth) val& = 1 - 2.5 MHz val& = 2 - 1.25 MHz val& = 3 - 625 KHz val& = 4 - 312.5 KHz val& = 5 - 156.3 KHz val& = 6 - 78.13 KHz val& = 7 - 39.06 KHz The maximum input frequency allowed to enter the counter will be 1/5 the filter frequency (approximate).
12	Unused.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CHAPTER 6. ANALOG INTERFACE

MODEL 1100 ONLY:

The Eason Technology Analog Interfaces allow the MODEL 1100 to monitor and stimulate devices which either generate or receive analog voltages. Three types of analog interface are available:

Option:	Capabilities:
A01	8 channel differential analog input, 12-bit A/D
A02	2 channel analog output, 12-bit D/A
A03	Combination 8 channel differential analog input; 2 channel analog output; 12-bit A/D and D/A

Refer to the MODEL 1100 MANUAL for more details on the hardware and specifications of the A01, A02, and A03.

CALL READAD Command

MODEL 1100 ONLY:

Purpose:

Read the specified analog input channel.

Syntax:

CALL READAD (*n,a*)

Comments:

n is the desired channel to be read (1 through 16).

a is the integer in which the result is to be placed.

Example:

The user connects +2 V to channel 1 (Vin to CH1+ and Gnd to CH1-)

```
CALL READAD(1,A)
PRINT A
410
Ready
```

CALL WRITEAD Statement

MODEL 1100 ONLY:

Purpose:

To write to the specified analog output channel.

Syntax:

CALL WRITEAD (*n, val*)

Comments:

n is the desired analog channel to be written to.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

val is the integer analog output value to be written. It is limited between +2047 (+10 V) and -2048 (-10V). The system has a transfer ratio of 4.883 MV per volt.

CALL RAMP Statement

Purpose:

To generate a an analog ramp on a designated A02 or A03 analog output. This ramp will have a specified starting and ending voltage, as well as an interval to ramp over.

Syntax:

CALL RAMP(***dac, start%, end%, increment&, time%*****)**

Comments:

dac specifies the desired D/A channel to update (1 through 4).

start% is the beginning value for the ramp in D/A counts.

end% is the ending value for the ramp in D/A counts.

increment& is a long integer which represents the D/A increment that the dac will change during 1 millisecond (multiplied by 2^16). It should be computed as follows:

I&=((endv%-startv%)/timev%)*65536)

Where: endv% is the ramp ending value startv% is the ramp starting value and timev% is the ramp execution time.

time% is the ramp execution time in milliseconds:

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CHAPTER 7. MODEL 1100 DIGITAL I/O INTERFACE

MODEL 1100 ONLY:

The Eason Technology MODEL 1100 expanded digital I/O options (options D24 and D48) allow the MODEL 1100's standard 24 digital I/O points to be supplemented with 24, 48, 72, or 96 additional digital I/O points. The additional I/O options have the same electrical characteristics as the internal 24 I/O. The only functional difference between the expanded I/O and the internal 24 I/O is the ability to generate interrupts. Only the internal 24 I/O are capable of driving interrupts.

The I/O is arranged in banks of 24 points. The internal I/O is Bank 0. The external I/O Banks are Banks 1-4. All commands that work with I/O (with the exception of the interrupt commands) act on the currently active Bank. At power up, the active Bank is Bank 0. Switching from one Bank to another is accomplished with the CALL IOBANK command.

CALL IOBANK Statement

MODEL 1100 ONLY:

Purpose:

To specify the I/O Bank that you wish to address with the various I/O commands (like OUTMAP, OUT, INP, etc.) Invoking the CALL IOBANK command makes the specified I/O Bank active for all commands following the call IOBANK command until a new CALL IOBANK command is issued.

Syntax:

CALL IOBANK (*bank*)

Comments:

bank specifies the active I/O Bank. I/O Bank 0 corresponds to the internal 24 I/O and is the default. I/O Banks 1-4 correspond to groups of 24 I/O found in the D24 and D48 Option modules. If you have one D24 Option module, you can access Banks 0 and 1. If you have one D48 Option module, you can access Banks 0, 1, and 2. If you have one D48 and one D24 Option module, you can access Banks 0, 1, 2, and 3. If you have two D48 Options, you can access banks 0, 1, 2, 3, and 4. The most recent CALL IOBANK command remains active until you explicitly change it.

I/O Bank	Minimum Options Needed	I/O Points
0	None	0-23
1	D48	24-47
2	D48	48-71
3	2 D48's	72-95
4	2 D48's	96-119

Example:

```
10 CALL IOBANK(1)
20 OUTMAP "1100"
30 CALL IOBANK(0)
40 INP A&
```

This simple program accesses the first Bank of the expanded I/O, sets bits 2 and 3 (I/O points 26 and 27) to 1 and sets bits 0 and 1 (I/O points 24 and 25) to zeros. It then changes to bank 0 (the internal 24 I/O) and inputs the status of I/O 0-23.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CHAPTER 8. 64K MEMORY OPTION (M02)

The M02 option adds an additional 32K bytes of storage to the 1000 SERIES PRODUCT's battery-backed up memory to bring the total memory up to 64K bytes. Larger programs and more data can be stored in this additional memory. In addition, nonvolatile storage registers can also be used when this option is added. 512 numeric registers, floating point, fixed precision, and integers (in any combination, note: what type is put in - must match - what type is taken out), as well as 16 string registers (128 bytes each) are available for storing seldom-changed program constants and data. Power failure, program loading, variable clearing, or gosub stack clearing will not affect the data stored in these registers.

You can tell if the M02 option has been installed in your unit by pressing the HELP key when the 1000 SERIES PRODUCT is not running a program (in command mode). By pressing the SYS key, the amount of memory installed will be indicated on the screen. It should read 64K. The amount of free memory will vary depending upon data and programs loaded. With no data or no programs loaded, the free memory should be 52735 bytes.

The M02 option adds access to the NVOL Variable as described below:

NVOL Variable

Purpose:

To allow numeric and string variables to be stored permanently. Once stored, they can be recalled after power failure, switching programs, clearing variables and clearing the gosub stack.

REQUIRES M02 option to work - 64K memory option.

Syntax:

NVOL[*type*](*n*) = 100
or:
A = NVOL[*type*](*n*)

Comments:

The optional specifier *type* can have the following characteristics:

% - specifies integer (16-bit) storage
& - extended precision integer (32-bit) storage
$ - string (127-byte) storage
! - floating point (32-bit) storage
none - floating point (default) storage

Think of the NVOL storage as two storage areas, one for strings and one for numeric variables. These storage areas can be treated like predimensioned arrays. *n* can have a range of 1 to 512 for numeric variables (of *type* %, &, or none). For strings (*type*: $), *n* can range from 1 to 16. Strings and numeric variables do not occupy the same space, and can be used simultaneously. For example:

```
100 NVOL(1) = 12.345
110 NVOL$(1) = "hello there"
```

The previous two program lines will save the numeric variable and the string in separate registers. They may be recalled and used as in the following example:

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
100 NVOL(1) = 12.345
110 NVOL$(1) = "HELLO THERE"
200 a = NVOL(1)
210 print a
220 print NVOL$(1)
run
 12.345
HELLO THERE
Ready
```

Numeric NVOL variables are stored according to the types they were specified when they are stored. NVOL%(1) will store register 1 as a 16-bit integer, whereas NVOL(1) will store it as a 32-bit floating point number. Storing register 1 as one numeric type and recalling it as another numeric type will get an unpredictable result. It is the programmer's responsibility to remember which types are assigned to each NVOL variable.

Again, string registers are stored in different locations; therefore, they do not conflict with the numeric registers. e.g. NVOL(1) and NVOL$(1) are two separate entities.

Trying to access the NVOL Variable with the standard M01 (32K) option installed, will result in a "**64K memory option required**" error. Using a value of *n* outside of the allowed ranges will result in a "**subscript error**".

ADDITIONAL STRING REGISTERS

Previously, the M02 option was limited to storing 16 separate 127 byte (character) strings. In some applications this has proven to be a detrimental limitation. For those applications where the event manager is not used, Eason Technology has added a larger and more flexible non-volatile string register space. This area allows you to use 4096 bytes of previously unavailable storage space for non-volatile string registers. This means that you can store 490 10 character strings, 81 50 character strings, 4096 1 character strings, etc. In addition, you still have access to the normal non-volatile registers in the M02 option (16 127 byte string and 512 floating point/long integer numeric registers) and those available in the M03 option. *Once again, if you use these extra string registers, you give up the event manager capability.*

CALL SETNVOLS Statement

Purpose:

Allow the programmer to set the size of a non-volatile string register allowed by the M02 option. By setting the size, the programmer can control the number of available string registers.

Syntax:

`CALL SETNVOLS(`*`var`*`)`

Comments:

var sets the maximum allowable size of the string for the nonvolatile string register. Sizes from 1 to 127 are allowed. By setting the size, the maximum number of string registers can be determined:

Maximum registers = 4096 / *var*

This command *MUST* be issued prior to using the `CALL RDNVOLS` and `CALL WRNVOLS` commands. It also must be issued after power up each time the program is run. Note that, when using this command, you will *not* erase the nonvolatile register space.

Example:

See the example at the end of the `CALL WRNVOLS` description.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CALL RDNVOLS Statement

Purpose:

To read string data from the adjustable length NVOL storage array.

Syntax:

CALL RDNVOLS(*addr, str$*)

Comments:

addr specifies the location or register number to read the string data from. The data is placed in ***str$***. The register number specified must also be less than or equal to the Maximum Registers as specified the `CALL SETNVOLS` command.

Example:

See the example at the end of the `CALL WRNVOLS` description.

CALL WRNVOLS Statement

Purpose:

To write string data into the adjustable length NVOL storage array.

Syntax:

CALL WRNVOLS(*addr, str$*)

Comments:

addr specifies the location or register number to write the string data to (***str$***). The string data must be shorter than or equal to the length specified by the `CALL SETNVOLS` Statement. The register number specified must also be less than or equal to the Maximum Registers as specified the `CALL SETNVOLS` command.

Example:

```
10 CALL SETNVOLS(10)
20 CALL WRNVOLS(1,"ABCDEFG")
30 CALL RDNVOLS(1,A$)
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CHAPTER 9. 128K MEMORY OPTION (M03)

The M03 option adds an additional 96K bytes of storage to the MODEL 1100's battery-backed up memory to bring the total memory up to 128K bytes. This option adds the M03 option (refer to Chapter 6), which increases the program storage area and adds its NVOL variable. The M03 option goes beyond the M02 option by adding an additional 64K bytes of Nonvolatile RAM memory to the Model 1100.

Before being used, this memory must first be initialized. Out of the 64K of memory in M03, 128 bytes are used for internal bookkeeping. The rest of the memory can be partitioned into storage space for non-volatile strings and numeric registers according to your needs. The CALL SETNVOL$(n) command allows you to set this partition, thereby allow the maximum memory flexibility in your application. You can configure up to 16864 numeric registers and 16 string registers, or 512 numeric registers and 527 string registers. By default, the M03 contains 256 strings and 8160 numeric registers, which can be modified using SETNVOL$ statement.

The CALL RDNVOL Statement reads data from the non-volatile registers. This works for memory which is placed in both the M02 and M03. M02 contains 16 strings and 512 numeric registers, while the allocation of the memory in the M03 is dependent upon the CALL SETNVOL statement.

The CALL WRNVOL Statement writes data to the Nonvolatile registers. There are four distinct commands, each allowing access to its own data type.

CALL SETNVOL$ Statement

MODEL 1100 ONLY:

Purpose:

To set the memory allocation of the M03 memory option. CALL SETNVOL$ performs this task by allocating string memory space.

REQUIRES M03 option to work - 128K memory option.

Syntax:

CALL SETNVOL$(*n*)

Comments:

n can be of the range 0 to 511.

Before being used, the memory allocated to the M03 option must first be initialized. Out of the 64K of memory in M03, 128 bytes are used for internal bookkeeping. The rest of the memory can be partitioned into storage space for non-volatile strings and numeric registers according to your needs. The CALL SETNVOL$(*n*) command allows you to set this partition, thereby allow the maximum memory flexibility in your application.

CALL SETNVOL$(*n*) sets up the number of strings allowed in the non-volatile memory. Where *n* is a number between 0 and 511. When you specify *n* to be 0 strings, you will have 16 strings (the number of string registers in the M02) and all of the remaining memory will be used as numeric registers. This results in 512+65408/4=16864 numeric registers (where 4 is the number of bytes allocated for EVERY register and 512 is the number of numeric registers in the M02 option). Similarly, specifying 511 strings will leave only the M02's numeric registers (512) and 527 string registers.

Every time you increase the number of string registers, the newly allocated strings will be emptied to nulls. When you deallocate the number of string registers, the numeric registers may contain garbage.

CALL RDNVOL Statement

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

MODEL 1100 ONLY:

Purpose:

To read the contents of the M03 Nonvolatile registers.

REQUIRES M03 option to work - 128K memory option.

Syntax:

CALL RDNVOL*(m,string var OR numeric var)*

Comments:

Where:

m is the number of the Nonvolatile register.

string var is a valid string variable or constant.

numeric var is a valid numeric variable or constant.

To read the non-volatile registers, use the CALL RDNVOL command. This works for memory which is placed in both the M02 and M03. M02 contains 16 strings and 512 numeric registers, while the allocation of the memory in the M03 is dependent upon the CALL SETNVOL$ statement.

By default, the M03 contains 256 strings and 8160 numeric registers, which can be modified using the CALL SETNVOL$ statement. When added to the M02 option (which must also be installed), you have a total of 272 string registers and 8672 numeric registers.

By using the CALL SETNVOL$ statement, you can have a maximum of 527 strings and 512 numeric registers (CALL SETNVOL$(511)), or 16 strings and 16352 numeric variables (CALL SETNVOL$(0)).

Numeric types must be carefully observed. There is no protection from saving a variable as a floating point and retrieving it as a integer. If you attempt this, you will receive garbage. Take care when saving or retrieving to match the numeric types.

Note: CALL RDNVOL provides another way to read to memory in M02. For example:

```
CALL RDNVOL(300,A!):PRINT A!
```

is identical to `PRINT NVOL!(300)`.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CALL WRNVOL Statement

MODEL 1100 ONLY:

Purpose:

To write data to the M03 Nonvolatile registers.

REQUIRES M03 option to work - 128K memory option.

Syntax:

CALL WRNVOL$(*m,string var*)
CALL WRNVOL!(*m,numeric var*)
CALL WRNVOL&(*m,numeric var*)
CALL WRNVOL%(*m,numeric var*)

Comments:

Where:

m is the number of the Nonvolatile register.
string var is a valid string variable or constant.
numeric var is a valid numeric variable or constant.

The Nonvolatile register addressing scheme is identical to the CALL RDNVOL Statement.

Numeric types must be carefully observed. There is no protection from saving a variable as a floating point and retrieving it as a integer. If you attempt this, you will receive garbage. Take care when saving or retrieving to match the numeric types.

Note: CALL WRNVOL provides another way to write to memory in M02. For example:

```
CALL WRNVOL$(1,"ABC")
```

is identical to NVOL$(1)="ABC".

```
CALL RDNVOL(300,A!):PRINT A!
```

is identical to PRINT NVOL!(300).

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CHAPTER 10. REAL TIME CLOCK OPTION (CLK)

The Real Time Clock Option (CLK Option) allows programs to access time of day, day, and date variables which are nonvolatile. Two commands allow BASIC to access the CLK Option. CALL RDCLOCK reads the contents of the CLK Option, and CALL WRCLOCK allows programs to write to the CLK Option. The CLK Option also has a the ability to generate time with a resolution of .01 seconds.

CALL RDCLOCK Statement

Purpose:

To read the date, day, and time from the clock.

Syntax:

CALL RDCLOCK (*date$,day,time$*)

Comments:

*date$* is a string variable which will contain the current date after the CALL. date$ will be returned in the following format:

MM/DD/YY
Where: MM is the month (1 through 12)
DD is the day (1 through 31)
YY is the year (88 through 21)

day is a numeric variable which will contain the current day of the week after the CALL. It has a range of 1 through 7. Where 1 is Sunday and 7 is Saturday.

time$ is a string variable which will contain the current time. It will be returned in the following format:

HH:MM:SS.SF
Where: HH is the hours (0 through 23)
MM is the minutes (0 through 59)
SS is the seconds (0 through 59)
SF is the fractional seconds (.00 through .99)

Example:

The following program will read the time and display it on screen continuously:

```
10 CLS
20 CALL RDCLOCK(date$,day,atime$)
30 POS 1,1
40 PRINT date$,day,atime$
50 GOTO 20
RUN
2/4/92              3                   13:30:13.25
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

CALL WRCLOCK Statement

Purpose:

To write the date, day, and time to the CLK option.

Syntax:

CALL WRCLOCK(*date$,day,time$*)

Comments:

*date$* is a string variable or string constant which contains the current date. date$ will must be in the following format:

MM/DD/YY
Where: MM is the month (1 through 12)
DD is the day (1 through 31)
YY is the year (88 through 21)

day is a numeric variable or constant which contains the current day of the week. It has a range of 1 through 7. Where 1 is Sunday and 7 is Saturday.

time$ is a string variable or string constant which contains the current time. It must be in the following format:

HH:MM:SS.SF
Where: HH is the hours (0 through 23)
MM is the minutes (0 through 59)
SS is the seconds (0 through 59)
SF is the fractional seconds (.00 through .99)

Example:

The following example sets the CLK Option and then reads it back:

```
10 CALL WRCLOCK("2/4/92",3,"12:15:31")
20 CALL RDCLOCK(date$,day,atime$)
30 PRINT date$,day,atime$
RUN
2/4/92              3                   12:15:31.25
Ready
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX A. ERROR CODES

ERROR CODE:	MEANING:	ERROR CODE:	MEANING:
1	syntax error	28	floating point overflow
2	variable required	29	integer overflow
3	out of string space	30	bad number
4	assignment '=' required	31	negative square root
5	line number required	32	negative or zero log
6	undefined line number	33	overflow in exp
7	line number overflow	34	overflow in power
8	illegal command	35	negative power
9	string overflow	36	illegal redefinition
10	illegal string size	37	undefined user function
11	illegal function	38	can't continue
12	illegal memory size	39	until without repeat
13	illegal edit	40	wend without while
14	device error	41	no wend statement found
15	dimension error	42	illegal loop nesting
16	subscript error	43	I/O error
17	next without for	44	illegal arguments
18	undefined array	45	system is locked
19	redimension error	46	Bank Error
20	gosub/return error	47	Timer not initialized
21	illegal error code	48	64K memory option required
22	out of memory	49	undefined label
23	division by zero	50	numeric variable required
24	bad data	51	internal error (call for Tech. Support)
25	out of data	52	128K memory option required
26	out of range	53	PLC communications not established
27	line length overflow	54	Invalid event data

1

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX B. KEY CODES

KEY	CODE
ENTER	13
SPACE	32
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
0	48
.	46
+	43

KEY	CODE
-	45
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77

KEY	CODE
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90
HELP	17408

KEY	CODE
INS	20992
F1	15104
F2	15360
F3	15616
F4	15872
F5	16128
F6	16384
F7	16640
F8	16896
F9	17152
UP	18432
DOWN	20480
LEFT	19200
RIGHT	19712
DEL	21248

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX C. ASCII CHARACTER CODES

Decimal Value	Hex Value	Character
000	00	NUL
001	01	SOH
002	02	STX
003	03	ETX
004	04	EOT
005	05	ENQ
006	06	ACK
007	07	BEL
008	08	BACKSPACE
009	09	TAB
010	0A	LINE FEED
011	0B	HOME
012	0C	FORM FEED
013	0D	CARRIAGE RET.
014	0E	SO
015	0F	SI
016	00	DLE
017	11	DC1
018	12	DC2
019	13	DC3
020	14	DC4
021	15	NAK
022	16	SYN
023	17	ETB
024	18	CAN
025	19	EM
026	1A	SUB
027	1B	ESCAPE
028	1C	FS
029	1D	GS
030	1E	RS
031	1F	US
032	20	(SPACE)
033	21	!
034	22	"
035	23	#
036	24	$
037	25	%
038	26	&
039	27	'
040	28	(
041	29	)
042	2A	*
043	2B	+
044	2C	,
045	2D	-
046	2E	.
047	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~

Decimal Characters 127-255 (7F-FF Hex) are not printable characters in the 1000 Series.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Artisan Technology Group

Artisan Technology Group is your source for quality new and certified-used/pre-owned equipment

- FAST SHIPPING AND DELIVERY
- TENS OF THOUSANDS OF IN-STOCK ITEMS
- EQUIPMENT DEMOS
- HUNDREDS OF MANUFACTURERS SUPPORTED
- LEASING/MONTHLY RENTALS
- ITAR CERTIFIED SECURE ASSET SOLUTIONS

SERVICE CENTER REPAIRS
Experienced engineers and technicians on staff at our full-service, in-house repair center

WE BUY USED EQUIPMENT
Sell your excess, underutilized, and idle used equipment
We also offer credit for buy-backs and trade-ins
www.artisantg.com/WeBuyEquipment ↗

InstraView™ **REMOTE INSPECTION**
Remotely inspect equipment before purchasing with our interactive website at ***www.instraview.com*** ↗

LOOKING FOR MORE INFORMATION?
Visit us on the web at ***www.artisantg.com*** ↗ for more information on price quotations, drivers, technical specifications, manuals, and documentation

Contact us: (888) 88-SOURCE | sales@artisantg.com | www.artisantg.com