Videx BASIC

for

DuraTrax®, LaserLite®, LaserLite Pro, and LaserLite Mx Data Collectors

Table of Contents

Introduction

This manual provides assistance to programmers and developers writing programs for the DuraTrax, LaserLite, LaserLite Pro, and LaserLite Mx data collectors using the BASIC programming language. The BASIC language implemented by Videx BASIC is a fairly large subset of Microsoft QuickBASIC with extensions for handling the DuraTrax, LaserLite, LaserLite Pro, and LaserLite Mx.

The programs are compiled to virtual machine language by a BASIC compiler (either a Windows program with drag-and-drop interface or a DOS command line compiler).

This manual consists of five chapters and three appendixes, containing a variety of aids for the developer.

Chapter 1 contains information on BASIC programming protocol.
Chapter 2 describes the BASIC data types.
Chapter 3 contains information on BASIC expressions and operators.
Chapter 4 describes the Videx BASIC statements and functions.
Chapter 5 contains information on the BASIC compiler programs.
The appendixes contain a list of the BASIC reserved words, information on the LaserLite Mx modulus, and two sample BASIC programs that can be used on the DuraTrax, LaserLite, LaserLite Pro, and LaserLite Mx data collectors.

Chapter 1

Videx BASIC

The Videx BASIC character set contains alphabetic characters, numeric characters, and special characters. The alphabetic characters are the letters A–Z, either uppercase or lowercase, and the underscore character. The numeric characters are the digits 0–9. The special characters and their purposes are:

Character	Purpose
<Enter>	Pressing the <Enter> key terminates input of a line
	Blank (or space)
$	Dollar sign (suffix for string data type)
%	Percent (suffix for integer data type)
'	Single quotation mark or apostrophe (comment indicator)
(	Left parenthesis
)	Right parenthesis
*	Asterisk (multiplication symbol)
+	Plus sign (addition symbol or concatenation operator)
,	Comma
-	Minus sign (subtraction symbol or prefix for negative number)
.	Period
/	Slash (division symbol)
:	Colon (separates statements on same line; ends line labels)
;	Semicolon
<	Less than
=	Equal sign (assignment symbol or relational operator)
>	Greater than
?	Question mark
|	Pipe (line continuation character)

Basic Program Line

The BASIC program line has the following syntax:

```
[line identifier] [statement] [: statement]...[comment]
```

Using Line Identifiers

Videx BASIC supports alphanumeric line labels for line identifiers.

An alphanumeric line label can be any combination of letters and digits, starting with a letter and ending with a colon. BASIC reserved words and type-declaration suffixes are not permitted. The following are valid alphanumeric line labels:

```
Gamma:
ShowList:
TestB5:
```

Case is not significant. The following line labels are equivalent:

```
gamma:
Gamma:
GAMMA:
```

Line labels may begin in any column, as long as they are the first characters other than blanks or tabs on the line. Blanks and tabs are allowed between an alphanumeric label and the colon following it. A line can have only one label.

BASIC Statements

A BASIC statement is either "executable" or "nonexecutable." An executable statement tells the program what to do next (for example, telling it to read input, write output, open a file, or take some other action). In contrast, a nonexecutable statement performs tasks such as allocating storage for variables, or declaring and defining variable types.

The following are nonexecutable BASIC statements:

- **REM** or ' (comment notation)
- **CONST** (defines constant)
- **DIM** (allocates storage)

A **comment** is a nonexecutable statement used to clarify a program's operation and purpose; it is introduced by either a **REM** statement or a single quote character ('). The following two lines are equivalent:

```
PRINT "Quantity remaining"    REM Print report label.
PRINT "Quantity remaining"     ' Print report label.
```

You may place more than one BASIC statement on a line, but a colon (:) must separate the statements, as illustrated below:

```
FOR I=1 TO 5 : PRINT "Welcome friends." : NEXT I
```

If the BASIC statement exceeds the character length of your screen, use the pipe (|) character at the end of the line to let the program know that the BASIC statement continues onto the next line. End the statement with the <Enter> key or with a colon character.

Every BASIC statement begins with a keyword and ends with either an end of the line (<Enter> key) or with a colon character. The one exception is assignment statements, where the initial keyword, **LET**, is optional.

Notes:

Chapter 2

Data Types

Every variable in BASIC has a data type that determines what can be stored in the variable. There are two categories of data in BASIC: string data and numeric data. Each category includes elementary data types. The following section summarizes these data types.

Elementary Data Types—String

All strings can have variable length up to a maximum size that is defined separately for each string. The default size of a string is 31 characters, but this can be changed to any value with the **DIM** statement (subject to Videx BASIC memory limitations).

Strings are terminated by the null character (ASCII 0). (Note: A consequence of this is that it is not possible to write the null character to a string or to a file with Videx BASIC. See the discussions on the **HEX$** function and **TOUCH** statement in Chapter 3 for more information on handling null characters in a string.)

Elementary Data Types—Numeric

• **Integer** (two bytes)

Integers are stored as sixteen-bit binary numbers ranging in value from -32,768 to 32,767.

Integer Numbers

All BASIC integers are represented as two's complement values. This is the most common way of representing integer numbers on a computer. Integers use 16 bits (2 bytes) of memory.

In two's complement representation, positive values are represented as straightforward binary numbers. For example, BASIC would store an integer value of 4 as the following sequence of 16 bits:

```
0000000000000100
```

Negative values are represented as the two's complement of the corresponding positive value. To form the two's complement (the negative) of the integer value 4, first take the representation above and change all the ones to zeros and all the zeros to ones:

```
1111111111111011
```

Then, add one to the result:

```
1111111111111100
```

The final result is how BASIC represents -4 as a binary number. Because of the way two's complement numbers are formed, every combination of bits representing a negative value has a 1 at the leftmost bit.

Constants

Constants are predefined values that do not change during program execution. There are two general types of constants: **literal constants** (such as numbers and strings) and **symbolic constants**.

Literal Constants

BASIC has two kinds of **literal constants**: **string** and **numeric**.

A **string constant** is a sequence of up to 32,767 alphanumeric characters enclosed by double quotation marks. These alphanumeric characters can be any of the characters (except the double quote character (") and carriage-return line-feed sequences) whose ASCII codes fall within the range of 1–255. This range includes both the actual ASCII characters (1–127) and the extended characters (128–255). The following are valid string constants:

```
"HELLO"
"$25,000.000"
"Number of Employees"
```

Also, any expression consisting only of other string constants, for example:

```
"HE" + "LLO"
```

Numeric constants are positive or negative integer numbers, consisting of one or more decimal digits (0–9), with a negative sign prefix (-) for negative numbers. The range for integer decimal constants is -32768 to 32767. Note: Numeric constants in BASIC cannot contain commas.

For example:

```
68
407
-1
29000
```

Symbolic Constants

BASIC provides **symbolic constants** that can be used in place of numeric or string values. The following fragment declares two symbolic constants and uses one to dimension an array:

```
CONST MAXCHAR%=254, MAXBUF%=MAXCHAR%+1
DIM Buffer% (MAXBUF%)
```

The name of a symbolic constant follows the same rules as a BASIC variable name. You may include a type-declaration character (**%** or **$**) in the name to indicate its type, but this character is not part of the name. For example, after the following declaration, the names **N$** and **N%** cannot be used for variable names because they have the same name as the constant:

```
CONST N=45
```

A constant's type is determined by an explicit type-declaration character or assumed to be an integer.

Variables

A variable is a name that refers to an object—a particular number or string. Simple variables refer to a single number or string. Array variables refer to a group of objects of the same type.

A numeric variable, whether simple or array, can only be assigned an integer numeric value; a string variable can only be assigned a character-string value. The variable must always match the type of data (numeric or string) assigned to it.

Variables can be assigned to a constant value:
```
A = 4
B$ = "sail the ocean blue "
```

Variables can also be assigned the value of another string or numeric variable:
```
A$ = B$
Profits = NetEarnings
```

Before a variable is assigned a value, its value is undefined. No assumptions should be made about the value of a variable until it has been explicitly assigned a value. See Chapter 3, "Expressions and Operators," for more information on the operators used in BASIC for combining variables and constants.

Variable Names

A BASIC variable name may contain any number of characters. The characters allowed in a variable name are letters, numbers, the underscore character, and the type-declaration characters (**%** and **$**). Variable names are not case sensitive in BASIC. The first character in a variable name must be a letter. See the section "Simple Variables" on page 12.

A variable name cannot be a reserved word, but embedded reserved words are allowed. For example, the following statement is illegal because **SEEK** is a reserved word (BASIC is not case sensitive):

```
Seek = 8
```

However, the following statement is legal, since the reserved word is embedded within the variable name:

```
TimeSeek = 8
```

Reserved words include all BASIC commands, statements, function names, and operator names. See Appendix A for a complete list of BASIC reserved words.

Variable Types

This section discusses the two variable types: **simple variables** (variables referring to a single object) and **array variables** (variables referring to a group of objects).

Simple Variables

Simple variables can be numeric or string variables. You can specify simple variable types by one of two ways:

- 1. Append one of the following type-declaration suffixes to the variable name:

    $ *(for string variables)*

    %   *(for integer variables)*

    The dollar sign (**$**) is the type-declaration character for string variables; it declares that the variable represents a string. You can assign a string constant of up to 31 characters to it by default, as in the following example:

    ```
    A$ = "SALES REPORT"
    ```

    The percent sign (**%**) is the type-declaration suffix for numeric variables; it declares that the variable represents an integer number.

    Variables without a type-declaration suffix are assumed to be integer variables.

- 2. Declare the variable in a **DIM** statement having the form:

    DIM *variablename*

    For example, the following statement declares the variable **A** as being a string type:

    ```
    DIM A$
    ```

Array Variables

An array is a group of objects that are referenced with the same variable name. The individual values in an array are called elements. These elements are the **array variables**, and they can be used in nearly any BASIC statement or function that uses variables. An array is "dimensioned" when you declare the name, type, and number of elements in the array.

Each array element is referred to by an array variable subscripted with an integer or an integer expression.

The maximum subscript value is set with the **DIM** statement. (See page 78–79 for more information on the **DIM** statement.)

You may have arrays of any variable type.

Array elements require a certain amount of memory, depending on the variable type. To find the approximate amount of memory required by an array, multiply the **number of elements** by the **bytes per element required** for the array type. For example, consider the following two arrays:

```
DIM Array1$ (7) * 12
DIM Array2% (99)
```

The first array, Array1, has 8 (0–7) string elements that can have up to 12 characters each, so Array1 takes approximately 96 (8 x 12) bytes of memory. The second array, Array2, has 100 (0–99) 2-byte integer elements so Array2 takes approximately 200 (100 x 2) bytes of memory.

Videx BASIC Memory Considerations

Videx BASIC supports up to 32K of data. Data includes all simple
variables, arrays, and constants declared in the program. It also includes
the name of the data file if it is a constant in the program.

For example, the following program line:

```
open "test.crf" for reference as #1
```

has two constants: **"test.crf"** and **1**.

The 32K data limit does not include run-time memory limitations, such
as file tables, data storage, or cross-reference files.

Videx BASIC also additionally supports up to 32K of code. Code is the
set of instructions to the BASIC engine in the DuraTrax, LaserLite,
LaserLite Pro, or LaserLite Mx operating system.

Chapter 3

Expressions & Operators

This chapter discusses how to combine, modify, compare, or get information about expressions by using the operators available in BASIC.

Anytime you do a calculation or manipulate a string, you are using expressions and operators. This chapter describes how expressions are formed, discusses the order in which BASIC uses operators, and concludes by describing the following four kinds of operators: **Arithmetic**, **Comparison**, **Bitwise**, and **String**.

Expressions and Operators

An expression can be a string or numeric constant, a variable, or a single value obtained by combining constants, variables, and other expressions with operators. Operators perform mathematical or logical operations on values. The operators provided by Videx BASIC can be divided into four categories, as follows:

1. **Arithmetic** operators perform calculations. See page 17 for information.

2. **Comparison** operators compare strings and numeric values. See page 19 for information.

3. **Bitwise** operators test complex conditions or manipulate individual bits. See pages 20–21 for information.

4. **String** operators combine and compare strings. See pages 22–23 for information.

Hierarchy of Operations

The BASIC operators have an order of precedence. Operations are executed in the following order:

1. **Negation** (-) and **NOT** (bitwise complement).

2. **Multiplication** (*), **division** (/), and **modulo arithmetic** (**MOD**).

3. **Addition** (+), **subtraction** (-), **AND** (bitwise conjunction), and **OR** (bitwise disjunction).

4. **Comparison** operations (=, >, <, <>, <=, >=).

Note: Operator precedence is somewhat different in Videx BASIC than in QuickBASIC; when in doubt, use parentheses.

If the operations are different and of the same level, the leftmost one is executed first, the rightmost last, as shown below:

```
A = 3 + 12 / 6 * 3 - 2   'A = 7
```

The order of operations would be as follows:

1. 12 / 6 (= 2)
2. 2 * 3 (= 6)
3. 3 + 6 (= 9)
4. 9 - 2 (= 7)

In a series of additions or multiplications, there is a fixed evaluation order, beginning with the first and preceding to the right. For example, in the following example:

```
B = 3 + 12 + 6 + 3 + 2   'B = 26
```

The order of operations would be as follows:

1. 3 + 12 (= 15)
2. 15 + 6 (= 21)
3. 21 + 3 (= 24)
4. 24 + 2 (= 26)

Arithmetic Operators

You can change the order in which the arithmetic operations are performed by using parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operation is maintained. Here are some sample algebraic expressions and their BASIC counterparts:

Algebraic Expression	BASIC Expression
$\dfrac{X-Y}{Z}$	$(X-Y)/Z$
$\dfrac{XY}{Z}$	$X*Y/Z$
$\dfrac{X+Y}{Z}$	$(X+Y)/Z$
$X(-Y)$	$X*(-Y)$ or $X*-Y$

Modulo Arithmetic

Modulo arithmetic is denoted by the modulus operator **MOD**. Modulo arithmetic provides the remainder of an integer division, rather than the quotient.

For example:

16 MOD 5: 16 ÷ 5 = 3 with a remainder of 1
Result = 1

3250 MOD 256: 3250 ÷ 256 = 12 with a remainder of 178
Result = 178

10 MOD 4: 10 ÷ 4 = 2 with a remainder of 2
Result = 2

68 MOD 8: 68 ÷ 8 = 8 with a remainder of 4
Result = 4

See **LOF** function, **SEEK** function, and **TOUCH** statement in Chapter 3 to see examples of how **MOD** is used.

• **Examples:**

```
PRINT 10 MOD 4    'Prints the remainder of 10 ÷ 4

PRINT 26 MOD 5    'Prints the remainder of 26 ÷ 5

PRINT 68 MOD 8    'Prints the remainder of 68 ÷ 8
```

• **Output:**

```
2

1

4
```

Comparison Operators

Comparison operators are used to compare two values, as shown in the following table. They work on both integers and strings. The result of the comparison is either true (nonzero) or false (zero). This result can then be used to make a decision regarding program flow. (Strings are compared lexicographically; see pages 22–23 for information on string comparisons.)

Comparison Operators and Their Functions

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 comparison operators are combined in one expression, the arithmetic operations are always done first. For example, the following expression is true if the value of **X + Y** is less than the value of **(T - 1)/Z**:

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

Bitwise Operators

Bitwise operators perform tests on multiple relations, bit manipulations, or Boolean operations; they return a true (nonzero) or false (zero) value to be used in making a decision.

Example:

```
IF (D < 200) AND (F < 4) THEN
WHILE (I > 10) OR (K < 0)
.
.
.
WEND
IF NOT P THEN PRINT "Name not found"
```

There are three bitwise operators in Videx BASIC: **NOT**, **AND**, and **OR**. They are described in the following table.

Videx BASIC Bitwise Operators

Operator	Meaning
NOT	Bitwise complement
AND	Bitwise conjunction
OR	Bitwise disjunction (inclusive "or")

Each operator returns results as indicated in the following table. A **T** indicates a true value (nonzero) and an **F** indicates a false value (zero).

Values Returned by Bitwise Operations

Values of		Value Returned by Bitwise Operator		
X	Y	NOT X	X AND Y	X OR Y
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

Bitwise operators compare each bit of the first operand with the corresponding bit in the second operand to compute the bit in the result.

In these bitwise comparisons, a 0 bit is equivalent to a false value (**F**) in the previous table, while a 1 bit is equivalent to a true value (**T**).

It is possible to use bitwise operators to test bytes for a particular bit pattern. For example, the **AND** operator can be used to mask all but one of the bits of a status byte, while the **OR** operator can be used to merge two bytes to create a particular binary value.

• **Examples:**

```
PRINT 63 AND 16
PRINT -1 AND 8
PRINT 10 OR 9
PRINT NOT 10, NOT 11, NOT 0    'NOT X = -(X + 1)
```

• **Output:**

```
16
8
11
-11    -12    -1
```

The first **PRINT** statement uses **AND** to combine **63** (111111 binary) and **16** (10000). When BASIC calculates the result of an **AND** it combines the numbers bit by bit, producing a 1 only when both bits are 1. Because the only bit that is a 1 in both numbers is the fifth bit, only the fifth bit in the result is a 1. The result is **16** or 10000 binary.

In the second **PRINT** statement, the numbers **-1** (1111111111111111 binary) and **8** (1000 binary) are combined using another **AND** operation. The only bit that is a 1 in both the numbers is the fourth bit, so the result is **8** decimal or 1000 binary.

The third **PRINT** statement uses an **OR** to combine **10** (binary 1010) and **9** (binary 1001). An **OR** produces a 1 bit whenever either bit is a 1, so the result of the **OR** in the third **PRINT** is **11** (binary 1011).

Performing a **NOT** on a number, as in the fourth **PRINT** statement, changes all one bits to zeros and all zero bits to ones. Because of the way two's complement numbers work, taking the **NOT** of a value is the same as adding 1 to the number and then negating the number. So in the final **PRINT** statement, the expression **NOT 10** gives a result of **-11**, **NOT 11** gives a result of **-12**, and **NOT 0** gives a result of **-1**.

String Operators

A string expression consists of string constants, string variables, and other string expressions combined by **string operators**. There are two classes of string operations: **concatenation** and **string comparison**.

Concatenation is the act of combining two strings. The plus sign (+) is the concatenation operator for strings. For example, the following program fragment combines the strings **CHR$(13)** and **CHR$(10)** to create a carriage return/line feed combination for use in a **PRINT** statement. Note: **CHR$(13)** represents a carriage return and **CHR$(10)** represents a line feed. The following program also combines variables **A$** and **B$** to produce the value **FILENAME**.

```
A$ = "FILE" : B$ = "NAME"
NL$ = CHR(13) + CHR$ (10)
AB$ = A$ + B$
PRINT AB$; NL$; "NEW ";AB$
```

• The output is: **FILENAME**
                  NEW FILENAME

Strings can be compared using the following comparison operators:

Operator	Description
<>	Not equal to
=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Note: These are the same relational operators used with numbers.

String comparisons are made by taking corresponding characters from each string and comparing their ASCII codes. If the ASCII codes are the same for all the characters in both strings, the strings are equal.

If the ASCII codes differ, the lower code number precedes the higher. If the end of one string is reached during string comparison, the shorter string is smaller if they are equal up to that point. Leading and trailing blanks are significant.

Following are examples of true string expressions:

```
"AA" < "AB"
"FILENAME" = "FILE" + "NAME"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"          'where B$ = "8/12/85"
```

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

Notes:

Chapter 4

Statement & Function Reference

This chapter is a dictionary of the Videx BASIC statements and functions for the DuraTrax, LaserLite, LaserLite Pro, and LaserLite Mx data collectors.

Please note that the following statements and functions are not supported by the Macintosh version of the Videx BASIC compiler: **BIN** function, **CARDCMD** statement, **CARDSTATUS** function, **HEX$** function, and **TOUCH** statement.

Each entry follows the following format:

- **Action** - Summarizes what the statement or function does.
- **Syntax** - Shows the model syntax.
- **Parameters** - Describes the variables used in the syntax.
- **Remarks** - Descriptions of arguments, options, and use.
- **Returns** - Value returned (functions only).
- **Example** - An example using the function or statement line. Note: In the examples, the statements **CLS**, **SLEEP**, and **LOCATE** are used to control the data collector's display. They are for effect only and may be ignored if you wish. **SLEEP 0** halts program execution until a key is pressed.

Following is a summary of the symbols used to define the syntax of the Videx BASIC functions and statements program lines:

- Terminal symbols are **bold**. These are to be typed in "as is."
- Variables are in *italics*.
- Variables that are assigned return values are in *UPPERCASE ITALICS*.
- [ ] enclose optional items.
- { } enclose items to repeat 0 or more times.
- ( ) enclose groups of items.
- The pipe character (|) separates alternatives.

ABS Function

• Action

Returns the absolute value of a numeric expression.

• Syntax

result% = ABS (*intexp%* **)**

• Parameters

Argument	Description
intexp%	Any whole number from -32768 to 32767.

• Remarks

The absolute value function returns the unsigned magnitude of its argument. For example, **ABS (-1)** and **ABS (1)** are both **1**.

• Returns

The unsigned value of *intexp%* as an integer.

• Example

```
REM ABS() Function

DIM result%,intexp%
CLS
intexp% = -1
PRINT "Value of intexp% is:"
PRINT intexp%;
SLEEP 0              'pause screen until keypress
CLS                 'clear the screen
result% = ABS(intexp%)
PRINT "ABSolute is:"
PRINT result%;
SLEEP 0
```

Output:
```
ABSolute is:
1
```

ASC Function

• **Action**

Returns the ASCII code of the first character in the string.

• **Syntax**

result% = ASC (*strexp$* **)**

• **Parameters**

Argument	Description
strexp$	Any string expression.

• **Remarks**

See also **CHR$**. The **CHR$** function complements **ASC**.

• **Returns**

The ASCII code of the first character in *strexp$* as an integer.

• **Example**

```
REM ASC() Function

DIM strexp$,result%
FOR i = 1 TO LEN("VIDEX")
   strexp$ = MID$("VIDEX",i,1)
   result% = ASC(strexp$)
   CLS
   PRINT "ASCII for "; strexp$; " is:"
   PRINT result%;
   SLEEP 0                    'pause screen until keypress
NEXT i
```

BEEP Statement

• **Action**

Sounds the beeper.

• **Syntax**

BEEP

• **Parameters**

None

• **Remarks**

The **BEEP** statement makes a sound through the beeper. This statement is the same as: **SOUND 1028, 100**

See also **SOUND**.

• **Example**

```
REM BEEP Statement

CLS
PRINT "This is a beep!"
BEEP
SLEEP 0
CLS
PRINT "This is :"
PRINT "SOUND 1028,100";
SOUND 1028,100
SLEEP 0
```

BIN Function

• **Action**

Converts hexadecimal digits to the corresponding integer value. (Videx BASIC) (Note: This function is not supported by the Macintosh version of the Videx BASIC compiler.)

• **Syntax**

result% = BIN (*hex_string$*, *num_digits%* **)**

• **Parameters**

Argument	Description
hex_string$	String variable containing hexadecimal digits to process at the beginning of the string. A hexadecimal digit can be either uppercase or lowercase.
num_digits%	Number of characters to convert (0 to 4). If zero is specified, only the number of valid hexadecimal digits encountered (up to 4) will be converted.

• **Remarks**

The last character converted (moving left to right) is always the least-significant nibble (4 bits) of the result. The more significant characters are leftmost in the string. For example, consider the two source strings **FACES** and **3C2GT**, the respective results for various *num_digits%* values would be:

hex_string$	*num_digits%*	**result%**	
FACES	0	-1330	(FACE hex)
	1	15	(000F hex)
	2	250	(00FA hex)
	3	4012	(0FAC hex)
	4	-1330	(FACE hex)
3C2GT	0	962	(03C2 hex)
	1	3	(0003 hex)
	2	60	(003C hex)
	3	962	(03C2 hex)
	4	-1	(error - too few digits)

The compiler flags any out-of-range values for *num_digits%*. If the
parameter is supplied by a variable, any out-of-range values are forced
into range by the formula:

```
good  =  ( (bad - 1) MOD 4 ) + 1
```

For example, a value of 5 becomes 1 (see Example 1) and a value of 19
becomes 3 (see Example 2).

*Note: MOD arithmetic provides the remainder of an integer division,
rather than the quotient. For example: **16 MOD 5 = 1**: 16 ÷ 5 = 3 with a
remainder **1**; **3250 MOD 256 = 178**: 3250 ÷ 256 = 12 with a remainder
of **178**. See page 18 for complete information on MOD arithmetic.*

- Example 1
    good = ((**5** - 1) MOD 4) + 1
    good = (4 MOD 4) + 1
    good = (0) + 1
    good = 1

- Example 2
    good = ((**19** - 1) MOD 4) + 1
    good = (18 MOD 4) + 1
    good = (2) + 1
    good = 3

The **BIN** function is similar to the **ASC** function, except that it interprets
pairs of hexadecimal digit characters (two per byte) as an integer rather
than giving just the ASCII value of an individual character.

See also **ASC**, **HEX$**, and **TOUCH**.

• **Returns**

Returns the value of the first *num_digits%* hexadecimal digits in the
string. If *num_digits%* is not zero and there are fewer digits than
specified, or if one of the characters of the required number is not a valid
hexadecimal digit, the function returns (-1). A legitimate value of **FFFF**
can be verified by string comparison.

• Example

```
REM BIN()and HEX$ Functions

DIM hex_data$ * 4
DIM i%, number%

FOR i% = 1 to 10
    number% = i% * 50
    hex_data$ = HEX$ (number%, 0) 'convert to hex
    CLS
    PRINT str$ (number%); " in hex:"
    PRINT hex_data$;                'display result
    SLEEP 0
    number% = bin (hex_data$, 0)  'convert back to decimal
    CLS
    PRINT hex_data$; " as decimal:"
    PRINT str$ (number%);          'display result
    SLEEP 0
NEXT i%
END
```

CARDCMD Statement *(for LaserLite Mx only)*

• Action

Constructs and sends commands to the memory card processor, adding required overhead and integrity elements. (Videx BASIC) (Note: This statement is not supported by the Macintosh version of the Videx BASIC compiler.)

• Syntax

CARDCMD *cmd_letter*[**!**][**,** *param_1* [**,** *param_2* [**,** *param_3*]]]

or

CARDCMD *command_str$*

• Parameters

Argument	Description
cmd_letter	A single letter that signifies the command to the memory card processor. Case is not significant. CRC checking is always used for this form of the command. Follow the letter immediately (no space) with an exclamation mark for commands that may take some time; the system does not wait for the result and retrieves it only when the **CARDSTATUS** command is issued. The default is to retrieve the result immediately after issuing the command (before returning). Special case: To activate a hardware reset of the memory card processor, this parameter should be an integer constant or variable with the value (-1).
param_x	These are parameters appropriate to the command to be executed. There can be zero or one string parameter, and if present it must be the last parameter. All parameters can be constants, variables, or expressions. The integer parameters can be specified (for **CARDCMD** only) by a single letter. The value is then interpreted to be the ASCII value of the letter, with case being significant. A side effect of this feature is that any integer parameters cannot begin with a variable name consisting of a single letter.

or

Argument	Description
command_str$	Complete command to send to the memory card processor. Must be the only parameter. Programmer is responsible for using the correct syntax within the command string. The system adds CRC error checking if the first character of the string is lowercase.

• **Remarks**

This command constructs and sends a command to the memory card processor. Its ability to accept a variable number of parameters means that it can adapt to the variations in syntax of the command set. Also, since single letters are interpreted as the integer ASCII value of the letter, both mode and file-type letters can be specified exactly as they are in the actual memory card processor command. (See page 38 for information on file types, pages 39–40 for additional notes on the **CARDCMD** statement commands, and Appendix B for information on hash values.)

The command syntax also allows the programmer to construct commands manually. A single string parameter (that is, with no command code parameter preceding) is presumed to contain the entire command already constructed. The string is sent as is, except when the first character is lowercase; then the CRC bytes are calculated and added to the end of the string.

The command sends the number of parameters specified (up to the maximum of three), so you must be sure that the number of parameters matches the syntax for the command. The BASIC compiler insists that there is at least one parameter and that if there are three parameters beyond the command letter, that the last one is a string.

After **CARDCMD** is used to send a command to the memory card processor, **CARDSTATUS** must be used to retrieve the response. Any data returned by previous commands, and not yet retrieved by a **CARDSTATUS** function call, will be lost.

See also **CARDSTATUS** and **HEX$**.

CARDCMD Global Errors

If the LaserLite Mx operating system is unable to successfully execute
the **CARDCMD** statement, it sets the global error flag. You may use the
ERR() function to retrieve an error number. **CARDCMD** can issue the
following errors:

Error #	Description
- 1	Communications with the memory card timed out.
- 2	There was no memory card module detected at startup.
- 3	There was no memory card inserted in the module detected at startup.
- 4	The card inserted in the memory card module does not have a recognized format.
- 11	The memory card processor is asleep and cannot receive a command. Removing the memory card during operations can cause this.

• **Example**

See the **CARDSTATUS** examples.

CARDCMD Statement Commands

The following table gives a brief description of the commands that can
be used by the **CARDCMD** statement. These commands provide
communication to and from the LaserLite Mx's memory card. The
command set is described in detail on pages 39–57. Parameters within
{ } are required; parameters within *[ ]* are optional. The parameters must
be sent in the given order.

command_str$	Brief Description
A, *[hash value]***,** *{record}*	Add a new record to the memory card's open file. See page 39 for more information.
C, *{F/S/I}***,** *[status bytes]*	List or send the memory card's file management or status report. See pages 40–43 for more information.
D, *{file type}***,** *{filename}* **D,** *{file handle}* **D,** *{file handle}***,** *{file type}***,** *{filename}*	Delete a file from the memory card. Delete an existing file. Rename an existing file. See page 44 for more information.

command_str$	Brief Description
F, *[hash value]***,** *[key field]* **F**	Search for a record in the memory card's open file with the given key field and send it to the host. Search for next record with same hash value or key field. See page 45 for more information.
H, *[hash value]***,** *[key field]* **H**	Delete a record from the memory card's open file with the given key field. Delete next record with same hash value and key field. See page 46 for more information.
K, &1092	Remove all deleted files from the memory card. See page 47 for more information.
M, *{# of records/bytes}***,** *{F/R}* **M** **M, H**	Move the pointer within the memory card's open file. Show the current record. Delete the record at the move pointer. See pages 48–50 for more information.
N, *[param1]*	Format a new memory card or determine memory card ID. See page 51 for more information.
O, *[modulus]***,** *{file type}***,***{filename}* **O,** *{file handle}*	Open a new or existing file on the memory card. Open an existing file. See pages 52–53 for more information.
Q, *{file type}***,** *{filename}*	Calculate the CRC of a file and send it back to the host. See page 54 for more information.
S, *{field bytes}***,** *{F/R}***,** *{string}*	Perform a search within the memory card's open file. See page 55 for more information.
V	Read the memory card's program version. See page 56 for more information.
Y	Repeat the last status byte or data. See page 57 for more information.
Z	Puts the data collector to sleep. See page 57 for more information.

File Types

The LaserLite Mx memory card system supports the following types of files: *identification* (**D**), *boot* (**B**), *sequential* (**S**), and *indexed* (**I/H**). Each file type serves a different purpose. The file types are described in the following table. See the *LaserLite Mx Developer's Reference Manual* for complete information.

Type	Purpose	Comments
D	A binary file that provides an ID for the memory card.	Only one (1) ID file may exist on the memory card at a time.
B	Two types of boot files serve two different purposes. One (CRD) is for the memory card operating software that executes in its 32K XRAM; the other (CPU) is the main operating system (***LMXxxx.OS***) and application of the LaserLite Mx.	These filenames are reserved by the LaserLite Mx system. CRD – The operating software for LaserLite Mx 32K XRAM. CPU – An operating system and application that may be booted from the memory card. Boot files must be appended with two bytes as bit complements to create a CRC of B001 hex over the entire file.
S	A binary file not intended for booting.	
I	This is the primary file type for data and cross-reference. Data is appended record-by-record. Each record is passed to the data management system to 'hash' and add to the record. This enables quick lookup.	A hash table size must be indicated when indexed files are created. Please see the notes about hash tables in Appendix B.
H	Another type of indexed file, but the decision of table entry is determined outside of the memory card data management system. This system may be useful for programs that must maintain a strict order of data, but require the ability to change or edit data.	**Vxcom** does not support transferring **H** files.

Notes on CARDCMD Statement Commands

- All communications between the host and the memory card must begin with a key character command and end with a carriage return.

- The key character of the commands may be lowercase or uppercase. When the key character is uppercase, the memory card module software is in debug mode; when the key character is lowercase, the software is in normal mode. The difference between debug mode and normal mode is that the communications between the host and memory card are CRC checked for normal mode, and not CRC checked for debug mode.

- For a lowercase key character command, the CRC is calculated from all of the bytes (including space and escape characters: **&** and **'** or **"**) before the carriage return. The complements of the two CRC bytes are sent at the end of the command, before the carriage return, with the low byte first. The CRC of all the bytes sent across before the carriage return will always be **&B001**, if correct. The CRC of the returns from the memory card will only include the bytes before the carriage return.

- There are two different types of fields besides the key character for a command: parameters to control the behavior of the command and data to be passed between the host and the memory card.
    - All of the fields for a command must be separated by a space and must be input in the order defined.
    - The parameters may be binary values, single byte ASCII, and ASCII strings (such as filename and key field).
    - A binary parameter may be up to four bytes long and can be input in both decimal and hexadecimal formats.
    - The data may be hexadecimal values (for binary data) or ASCII strings (for records).
    - All hexadecimal values used in the command line must begin with an ampersand (&) and end with an ampersand, space, or carriage return.
    - All ASCII strings (with more than one character) must be enclosed in either single ('*xxx*') or double ("*xxx*") quotation marks; a single non-numeric ASCII byte does not need to be enclosed in quotation marks.

- The returns from the memory card can be status bytes (in decimal form), data (in hexadecimal or ASCII string format), or prompts responding to a single carriage input. The prompt for the firmware includes three bytes: a carriage return, a line feed, and a semicolon (:). The prompt for the memory card's boot program (Data Management System (DMS)) is: a carriage return, line feed, and greater-than sign (>). All hexadecimal values returned by the memory card will begin with an ampersand (&).

- For indexed files, the DMS handles one record at a time. A tab character must separate the fields within the record.

- For sequential files, every byte in the data field of a command is written to the file.

- Doubling the "escape character" forces the character to be a data byte, for example, **"&&"** in a quoted string represents character **&**; **"''"** represents a **'** character. If two continuous escape characters need to be used as Escape (as when sending two strings together) a space must be used to separate the strings.

- Do not include a carriage return or a backspace in a quoted string. A carriage return can only be used as the end of transmission mark and the backspace key will delete the previous byte. If nonprintable bytes need to be sent as part of a string, they should be sent as a hexadecimal value.

- Any binary data except the control characters (08 hex, 0D hex) and the escape characters (&, " or ') can be sent as ASCII strings.

CARDCMD Statement Commands

The following sections provide complete information about each **CARDCMD** statement command.

A, *[hash value]***,** *{record}*

Adds data to the memory card's open file (see the **O** command on pages 52–53 to open file). For an indexed (**I**) file, the card module first calculates the hash entry based on the key field of the record and then looks up the hash table for a pointer. If no pointer is found, it is the first record for the entry. The *record* is saved as the current end-of-file and the pointer at the hash table is set. If a pointer is found, a collision occurred for this entry. The DMS traverses the list and finds the last record with the same *hash value*. The new record is saved as the current end-of-file and the pointer is set at the last record.

For type **I** files, the *hash value* is calculated internally. You can have more than one record with the same key field. The key field is delimited from other fields with a tab character (09 hex). For type **H** files, the *hash value* must also be passed with the command. For type **S/D/B** files, the **A** command appends *{record}* as data.

Example: Add a record to the open file:

```
CARDCMD A, "This is a record"
```

Successful return values:

> 00   (Record added)

Possible error codes that can be returned from this command:

> 01, 03, 04, 10, 32, 34, 39, 40, 41, 42, 53, 61, 63 *(See pages 59–60 for a description of the error codes.)*

CARDCMD Statement Commands - continued

C, *{F/S/I}*, *[status bytes]*

Lists the memory card's file management or status information or sends a setup configuration to the memory card processor.

>   *F* - List file management report.
>   *S* - List status report.
>   *I* - Send status report to the memory card processor.

F - List File Management Report

The file management report is sent out first, if asked, followed by individual file information. The filenames are enclosed in quotation marks. The file type is sent out as hexadecimal; all hexadecimal values in the report start with an ampersand (&). A space is used to separate hexadecimal from ASCII, or different parts of the listing (for example, a space between file 1 and file 2). The following tables show the order of the data format. A sample of the file management report is shown on the following page.

File Management Report Format:

bytes 1–2	Total space of the card (Kbytes).
bytes 3–4	Bad space (Kbytes).
bytes 5–6	Available space (Kbytes).
bytes 7	Number of files (including deleted files).
bytes 8–b1	1st valid file information.
bytes b2–b3	2nd valid file information.
bytes b4–b5	3rd valid file information.
...	

Individual File Information:

byte 1	File serial number in the memory card.
byte 2	File status.
byte 3	File type.
bytes 4–5	File table size.
bytes 6–7	The file size (Kbytes).
bytes 8–16	The filename (variable length from 1–18 bytes).

Example: Request file management report:

```
CARDCMD C, F
```

Example Return:

```
&07980000076805 &01FF44FFFF0004 "card1"
&02FF42FFFF0014 "CRD" &030049089B000C
"data.txt" &03FF49089B000C "olddata.txt"
&04FF49089B000C "data.txt"
```

Note: The **F** command in a commands file issues the **C, F** command to the memory card. It creates the following report on the information above:

Memory Card File and Memory Status Report

```
Total Space:          1944 Kbytes
Bad Space:               0 Kbytes
Available Space:      1896 Kbytes
Retrievable Space:      12 Kbytes

Number of deleted files:   1
Number of valid files:     4
```

File Number	Deleted	Type	Table Size	Kbytes	Name
1	No	D	n/a	4	"card1"
2	No	B	n/a	20	"CRD"
3	Yes	I	2203	12	"data.txt"
3	No	I	2203	12	"olddata.txt"
4	No	I	2203	12	"data.txt"

CARDCMD Statement Commands – continued

The LaserLite Mx only allows one open file at a time. When you open a file, the Mx Data Management System (DMS) automatically closes a previously opened file and loses its reference to any given record. If your application requires you to open and close multiple files and you wish to re-open a file and return to the same record, then you must use the **Cardcmd C, S** and **Cardcmd C, I** statements.

S – List Status Report

At any given time the LaserLite Mx may have an open file and a pointer to data or to a record within that file. When power is removed from the memory card module during sleep, the data management system loses track of this information. The status report contains the information needed to restore the memory card system to a previous state. The operating system handles this automatically, but a developer may want to maintain a reference to a file and record among multiple files. This command provides that capability.

Status Report Format:

byte 1	Last command before the **C** command.
bytes 2–6	The move pointer: block # (2 bytes), page number (1 byte), column address (2 bytes).
byte 7	File type.
bytes 8–b1	Open filename (variable length).

Example: Request status report:

```
Cardcmd C, S
```

Example Return:

```
&4F001503006A04
```

Example Program:

```
Cardcmd C, S          'Issue statement to get current
                      'file and record pointer
Gosub process_cardcmd_error       'Process any errors
Result% = Cardstatus(file1$)'Get status in file1$
Gosub process_card_error   'Process any return errors
Cardcmd O, I, "file2.txt"  'Open the next file
```

I - Send Status Report to Memory Card Processor for Update

The status string retrieved by the **C, S** command may be sent to the memory card processor with the **C, I** command. The LaserLite Mx operating system and BASIC language automatically update the state of the memory card in its sleep/wake routines. Normally, you will not be concerned with updating the status, except when developing applications that use multiple files.

Example Program:

```
Cardcmd C, I, file1$    'Re-open previously opened file and
                        'restore pointer to specific record
Gosub process_cardcmd_error      'Process any errors
Result% = Cardstatus(file1$)     'Get status in file1$
Gosub process_card_error   'Process any errors from return
```

Successful return values:

> File management report (for **C, F**); status report (for **C, S**); 00 (for **C, I**).

Possible error codes that can be returned from this command:

> 01, 03, 04, 10, 31, 32, 51, 52, 63 *(See pages 59–60 for a description of the error codes.)*

CARDCMD Statement Commands – continued

D, *[modulus]*, *{file type}*, *{filename}*

> *or*

D, *{file handle}*

> *or*

D, *{file handle}*, *{file type}*, *{new name}*

Deletes or renames a memory card file. To delete a file, list the *file type* (indexed (**I** or **H**), sequential (**S**), identification (**D**), or boot (**B**)) and *filename* of the file to delete. The *filename* can have up to 18 bytes; or you can use the *file handle* number (if known from a previous **O** command).

To rename a file, you must refer to the file by the *file handle* number and *file type* and pass it the new *filename*.

Example 1: Delete an indexed file with *filename* **data.txt**:

```
Cardcmd D, I, "data.txt"
```

Example 2: Rename an indexed file with *filename* **data.txt** and *file handle* **03** to **olddata.txt**:

```
Cardcmd D, 03, I, "olddata.txt"
```

Successful return values:

> 00   (File deleted or renamed)

Possible error codes that can be returned from this command:

> 01, 03, 04, 09, 10, 31, 33, 40, 63, 65 *(See pages 59–60 for a description of the error codes.)*

F, *[hash value]***,** *[key field]*

> *or*

F

Searches for a record with the given *key field* in the memory card's open file and sends it to the host. This command only functions with file types **I** and **H**. When a *key field* is not provided, and the last command was an **H** or an **F** command with a *key field*, it tries to read the next record with the same *key field* as was used in the last **H** command.

For example, to read multiple records with the same key field, the following commands can be used. Assuming there are three records with the *key field* **ABCDEFG** in an indexed file, then

> **Cardcmd F, "ABCDEFG"**: reads the first record;
>
> **Cardcmd F**: reads the second record;
>
> **Cardcmd H**: deletes the third record.

To read all of the records in an indexed file with the same *hash value*, the following commands can be used:

> **Cardcmd F,** *[key field]*: first record in the chain;
>
> **Cardcmd F**: second record in the chain.

Example: Find a record in the open file:

> **Cardcmd F, "abc"**

Return:

> **"abc"** *<field2>*   *<field3>*   Record found.
>
> **35**   No records match the find request.

Successful return values:

> A record.

Possible error codes that can be returned from this command:

> 01, 03, 04, 06, 08, 10, 32, 35, 53, 63 *(See pages 59–60 for a description of the error codes.)*

CARDCMD Statement Commands – continued

H, *[hash value]***,** *[key field]*

Deletes a record with the given *key field* from the memory card's open file. This command only functions with file types **I** and **H**. For an indexed file, the program goes through the hash table, finds the record, and sets the record status bit to zero. It does not change the pointer or erase the record. When no *key field* is provided, and the last command is an **F** or an **H** command with a *key field*, it tries to delete the next record with the same *key field* as in the last **H** command.

Example: Delete a record from the open file:

```
Cardcmd H, "abc"
```

Successful return values:

> 00   (Record deleted)

Possible error codes that can be returned from this command:

> 01, 03, 04, 06, 08, 10, 32, 35, 39, 40, 53, 63, 65 *(See pages 59–60 for a description of the error codes.)*

K, &1092

Removes deleted files from the memory card. This command copies the file management report from the control blocks to other blocks, changes content, erases the control blocks, and copies the information back to the control blocks. Do not interrupt the processor during the operation (power down or reset) as it is possible that file management information can be lost. This command should only be used <u>after</u> important data is transferred from the memory card to the computer.

The **&1092** parameter is required and allows the command to remove the deleted files from the memory card.

Example: Clean up the card:

```
Cardcmd K, &1092
```

Successful return values:

00	(Deleted files removed from the memory card)

Possible error codes that can be returned from this command:

01, 03, 04, 10, 40, 61, 62, 63, 65, 67, 68 *(See pages 59–60 for a description of the error codes.)*

CARDCMD Statement Commands – continued

M, *{number of records/bytes}*, *{F/R}*

 or

M, H, *{file handle}*

 or

M

Moves the pointer within the memory card's open file (see the **O** command on pages 52–53 to open file). For indexed (**I** or **H**) files, this command moves the pointer forward (F) or backward (R) a certain *number of records* within the open file, and sends out the current record that the pointer is pointing to. When *number of records* = **&FFFF**, the pointer is moved to the end-of-file (F) or the beginning-of-file (R). For sequential (**S**) and identification (**D**) files, the number is in bytes, and the program sends out 256 bytes. When the *number of bytes* = **0** (or is omitted), the program sends out the current record or 256 bytes, but does not move the pointer.

When a file is opened, the move pointer is set at the end-of-file. An **F** or an **H** command sets the move pointer at that record.

Example 1: Move forward 128K records from current position in open file:

```
Cardcmd M, &020000, F
```

Returns:

 "abcdefg" *<field 2> <field 3> <…>*
         Record found; data reported.

 36   End-of-file encountered.

Example 2: Move to beginning of the open file:

```
Cardcmd M, &FFFF, R
```

Returns:

    `"abcdefg data"`   Record found, data reported.

    `23`   There is no data in the file.

Example 3: Delete the record at the move pointer:

```
Cardcmd M, H
```

Returns:

    `00`   Record deleted.

Successful return values:

A record or 256 bytes of binary data.

Possible error codes that can be returned from this command:

01, 03, 04, 10, 23, 36, 37, 39, 40, 53, 63, 65 *(See pages 59–60 for a description of the error codes.)*

CARDCMD Statement Commands – continued

The Move Pointer

The **M** card command moves a pointer within an open data file. The following table indicates the location of the move pointer under different circumstances.

Condition	Location for file types I/H (indexed files)	Location for file types B/D/S (sequential files)
Open a file	Last record.	Last 256 bytes.
Append record	Last record.	Last 256 bytes.
Scrolling	Current record.	Current 256 bytes.
Closed file	Invalid.	Invalid.
Open file with no data	"No data condition" (error 23).	"No data condition" (error 23).
Delete record with the **H** command or **M, H** command	At the deleted record. Must issue another **M** command before issuing an **M, H** command. The **M** command then moves to the next record in the data file. If there is not a valid next record, then the move pointer moves to the previous record.	Not applicable.
F command	At the found record or, if not found, at the bottom of the data file.	Not applicable.
S command	At the found record or, if not found, at the bottom of the data file.	Not applicable.

N, *[param1]*

Formats the memory card. **N**, by itself, closes any open files and restarts the memory card program. It also locks some of the physical functions, so the IRAM, XRAM, and memory card contents cannot be accidentally changed using low-level routines. To return the ID of the memory card, issue the **N** command without a parameter.

The parameter **&0129** unlocks low-level physical functions permitting writing directly to the card module's XRAM memory and formatting the memory card.

The parameter **&1092** erases the entire card and formats the card, if it is not formatted.

Example 1: Format memory card:

```
Cardcmd N, &1092
```

Returns:

> **00**   Memory card formatted.

Example 2: Determine memory card ID:

```
Cardcmd N
```

Returns:

> **"092612456"**   The card ID name, card module formatted.

Successful return values:

> ID of the memory card.

Possible error codes that can be returned from this command:

> 01, 03, 04, 07, 10, 22, 63, 65 *(See pages 59–60 for a description of the error codes.)*

CARDCMD Statement Commands – continued

O, *[modulus]***,** *{file type}***,** *{filename}*

> *or*

O, *{file handle}*

Opens an existing or a new file on the memory card. To open an existing file, only the *file type* (indexed (**I** or **H**), sequential (**S**), identification (**D**), or boot (**B**)) and the *filename* is needed; or you can use the *file handle* number, if known from a previous **O** command. The *modulus* (number of hash table entries) can be zero. The *modulus* can be omitted for sequential, identification, or boot files; for indexed or hashed files the *modulus* can be from 1–65535. (Note: See Appendix B for information on the modulus.) The *filename* can have up to 18 bytes. When a file is opened, any following operations are directed to the open file; any previously opened files are closed. When the module is powered up, the firmware boots the memory card and starts to run the DMS program from the XRAM. If successful, the **O** command returns the *file handle* number of the opened file.

One memory card ID file can be opened per card. The *filename* is used as the card ID, and any contents can be written to the file as a sequential file. The ID can be found by writing an ID file to the memory card with any name. It can also be found by issuing an **N** command (see page 51) without a parameter.

Example 1: Open an indexed file with table size 2203 and filename **test1.txt**:

```
Cardcmd O, 2203, I, "test1.txt"
```

or

```
Cardcmd O, &089B, I, "test1.txt"
```

or

```
Cardcmd O, 03
```

Returns:

       &03   File opened successfully and **03** hex is the file handle.

       39   Memory full.

Example 2: Open the ID file with the filename **Videx**:

```
Cardcmd O, D, "Videx"
```

Returns:

&01	ID file created successfully, with 'Videx' as the card ID and assigned to file handle 01.
"092612456"	ID file could not be created because an ID file already exists with '092612456' as the real card ID.

Successful return values:

File handle number or card ID name.

Possible error codes that can be returned from this command:

01, 03, 04, 07, 10, 31, 33, 39, 40, 51, 52, 63 *(See pages 59–60 for a description of the error codes.)*

CARDCMD Statement Commands – continued

Q, *{file type}*, *{filename}*

Calculates the CRC of a file and sends it back to the host. This command is useful when sending a file to the memory card from the computer. The **Vxcom** and **Download** communications programs automatically call this command when sending files to the memory card.

Example: Check the CRC of a boot file:

```
CARDCMD Q, B, "CPU"
```

Returns:

> **&B001** The 16-bit CRC of the boot file (boot files on the LaserLite Mx system must always return a CRC of **&B001**).

Successful return values:

> 16 bit CRC of the file.

Possible error codes that can be returned from this command:

> 01, 03, 04, 10, 31, 32, 51, 52, 63 *(See pages 59–60 for a description of the error codes.)*

S, *{field number/bytes}***,** *{F/R}***,** *{string}*

Performs a search within the memory card's open file. For an indexed file (**I/H**), this command searches for a record with the given string within the given number of records. The field number limits the search at the particular field. A zero field number forces the program to search through the entire field for the given string. The key field is the number one field. The field number is the fourth byte of the first parameter and the number of records that were searched is given as the three least significant bytes of the first parameter.

For a binary file, this command searches through the *number of bytes* trying to match the given *string*. With no parameter, it reads the last record or 256 bytes found. This command uses the same pointer as the **M** command. Since this command does not use indexing, there may be some noticeable delay when using this command to search through large amounts of data.

Example: Search the open file for a record in field two with the pattern "abcdefg" within the next 129838 records from the current move pointer:

        CARDCMD S, &0201FB2E, F, "abcdefg"

Returns:

        "abcdefghijklmnop"   Record found.

Successful return values:

        A record or 256 bytes of binary data containing the search
        string.

Possible error codes that can be returned from this command:

        01, 03, 04, 10, 23, 35, 36, 37, 39, 40, 63, 65 *(See pages 59–60
        for a description of the error codes.)*

V

Reads the memory card's program version. The version is sent out in ASCII format. The version number for the DMS is:

    VTDMSx.xx

The version number of the firmware is:

    VTMCFx.xx

Example: Read the DMS version:

    CARDCMD V

Returns:

> **"VTDMS1.02"**   The DMS version.

Successful return values:

> Version number of the DMS or firmware.

Possible error codes that can be returned from this command:

> 03, 04, 10 *(See pages 59–60 for a description of the error codes.)*

Y

Repeats the data sent by the memory card processor.

Example: If the last command was the **V** example previously described:

   CARDCMD Y

Returns:

   VTDMS1.02   The DMS version.

Successful return values:

   Last data.

Possible error codes that can be returned from this command:

   03, 04, 10 *(See pages 59–60 for a description of the error codes.)*

Z

Puts the memory card processor in sleep mode.

Example: Put the memory card processor to sleep:

   CARDCMD Z

Returns:

   00   LaserLite Mx memory card processor is in sleep mode.

Successful return values:

   00   (Memory card processor in sleep mode)

Possible error codes that can be returned from this command:

   03, 04, 10 *(See pages 59–60 for a description of the error codes.)*

CARDSTATUS Function *(for LaserLite Mx only)*

• Action

Retrieves the response from the memory card processor after it is given a command. (Videx BASIC) (Note: This function is not supported by the Macintosh version of the Videx BASIC compiler.)

• Syntax

result% = CARDSTATUS (*BUFFER_VAR$* **)**

• Parameters

Argument	Description
BUFFER_VAR$	String variable that receives a data component (if there is any) of the memory card processor response. Do not use a string constant or array variable element. The dimensioned size of the string variable must be able to hold any data response that might result from the command given. If it is too small, the data is truncated.

• Remarks

If the data is in the form of a single-quoted ASCII field, the enclosing single quotes are removed and any embedded single quotes are interpreted. Hexadecimal fields are passed along unchanged; that is, the return string contains the hexadecimal digits with the initial ampersand (&) marker and terminating marker (if it is not a carriage return).

Whenever the **CARDCMD** statement is executed, the values returned by this function are cleared even if they have not been retrieved. See also **CARDCMD** and **BIN**.

• Returns

If the operation is successful, **CARDSTATUS** returns zero and the data returned by the memory card processor is placed in *BUFFER_VAR$*.

If an error is detected by the operating system, **CARDSTATUS** returns a negative number. If the memory card processor times out, the function returns **-1**. If there is no memory card module, the function returns **-2**. If a memory card was not installed in the card module at last startup from sleep, the function returns **-3**. See the following table for information on other error values.

CARDSTATUS Return Values (returned by Operating System)

Return Value	Description
- 1	Communications with the memory card timed out.
- 2	There was no memory card module detected at startup.
- 3	There was no memory card inserted in the module detected at startup.
- 5	The CRC of the return from the memory card system does not match.
- 6	The return from the memory card system is unrecognized.
- 7	The designated string is too short for return from the memory card.
- 8	No card command was issued since the last call to **CARDSTATUS**.
- 9	**CARDCMD** statement resulted in an error; get from **ERR** ().
- 10	Card processor is still busy when **CARDSTATUS** called.
- 11	Card is asleep when **CARDSTATUS** called.
- 15	The return from the memory card has exceeded the 2K buffer.

If communication occurs with the memory module, but an error or warning condition is detected, **CARDSTATUS** returns a positive number. The following table (continued on the next page) lists these return values:

CARDSTATUS Return Values (returned by Memory Module)

Return Value	Vxcom Error Code	Description
00	21000	Operation successful.
01	21001	Unrecognized card. This version recognizes Toshiba's SSFDC 2 MB, 4 MB, and 8 MB memory cards.
02	21002	Unrecognized card.
03	21003	Syntax error.
04	21004	CRC of command did not match.
05	21005	Unknown command.
06	21006	Missing parameters for this command.
07	21007	Incorrect parameters for this command.
08	21008	Binary file.
09	21009	Incorrect file type.
10	21010	Too much data in the command.

Return Value	Vxcom Error Code	Description
22	21022	No ID file.
23	21023	No data.
31	21031	No such file exists.
32	21032	No file opened.
33	21033	Too many files (>60 files).
34	21034	The page has already been written to four times. (Note: Toshiba only allows you to write to a page four times.)
35	21035	No such record exists.
36	21036	End of file.
37	21037	Beginning of file.
38	21038	Time-out occurred.
39	21039	Memory full.
40	21040	Write/protect encountered.
41	21041	Record too large (>1024 bytes).
42	21042	Field too large (>255 bytes).
43	21043	Some of the physical functions are locked to avoid data corruption. Try: **N, &0129** to unlock.
51	21051	File management error (control data was changed).
52	21052	Sequential file corrupted (by accidental power failure).
53	21053	Data corrupted.
54	21054	CRC of boot file did not match (program data may be corrupted).
55	21055	The bootcard program is too large (must be less than 24 KB).
61	21061	Card program failed (card may be worn out).
62	21062	Block erase failed (card may be worn out).
63	21063	Card format is incorrect.
64, 65	21064, 21065	First block of memory is bad (card may be damaged).
66	21066	Too many bad blocks (card may be damaged).
67	21067	Most of the reserved control blocks are bad (card worn out).
68	21068	The card has been erased more than 32,768 times (each **K, &1092** or **N, &1092** counts as one erasing).

• Example 1

This routine opens a file on the memory card called "data.txt." The
process_cardcmd_error routine checks the global error flag and
processes the error if there is one. If there is no error, then the routine
calls **CARDSTATUS** to get the return value, again processing any
errors if there are any. The subroutines referred to in the following
example are located in the sample MXDEMO program located in
Appendix C.

```
cardcmd O, 2203, I, "data.txt"
gosub process_cardcmd_error
'report errors from sending command to card
IF crderr% = 0 THEN
      cardresult% = cardstatus(cardreturn$)
                                      'retrieve results
      gosub process_card_error      'handle any errors
ENDIF
```

• Example 2

This routine scrolls down in an open data file. In the demo programs,
INPUTEVT calls this routine in response to the user pressing the down-
arrow key. The subroutines referred to in the following example are
located in the sample MXDEMO program located in Appendix C.

```
fn_down:

   sound 1397, 10 'click so user knows we got the key
   if (mode% = mode_top%) then
'we are at the top of the file, so don't move pointer
      cardcmd M
      gosub process_cardcmd_error
      cardresult% = cardstatus(cardreturn$)
      gosub process_card_error
'get the next line (if not at the top of the file)

   else
      cardcmd M, 1, F
      gosub process_cardcmd_error
      cardresult% = cardstatus(cardreturn$)
      gosub process_card_error
   endif

   if cardresult% = 36 then
      gosub scroll_info
'if at end of file, display system info
      mode = mode_bottom%

   elseif cardresult% = 23 then
      gosub scroll_info
      mode = mode_bottom%

   else
      display$ = cardreturn$
      mode% = mode_dn%
   endif

   full_display$ = display$
'initialize for scroll right & left
   shift% = 0

return
```

CHR$ Function

• **Action**

Returns a one-character string whose ASCII code is the argument.

• **Syntax**

result$ = CHR$ (*code%***)**

• **Parameters**

Argument	Description
code%	ASCII code (0–255) of character.

• **Remarks**

CHR$ is commonly used to send a special character to the display. The tables on the following two pages list most of the display characters and their ASCII code. There are other characters the LCD can display; feel free to experiment. If you enter a character that the LCD cannot interpret, it is displayed as a blank.

See also **OPTION** statement *(options 512–575)* and **ASC**. The **ASC** function complements **CHR$**.

• **Returns**

The character that the ASCII code represents.

• Example

This routine shows how **CHR$** can be used to include a double quote (") in a string.

```
REM CHR$() Function
REM This routine uses CHR$ to include a double quote (")
REM in a string.
Dim quote$
CLS
quote$ = CHR$ (34)
PRINT "This is a :"
PRINT quote$; "Quoted String"; quote$;
SLEEP 0
```

Output:
```
This is a :"Quoted String"
```

Table of Display Characters and Corresponding ASCII Codes

Character	ASCII	Character	ASCII	Character	ASCII
Line feed	10	,	44	:	58
Carriage return	13	-	45	;	59
Space	32	.	46	<	60
!	33	/	47	=	61
"	34	0	48	>	62
#	35	1	49	?	63
$	36	2	50	@	64
%	37	3	51	A	65
&	38	4	52	B	66
'	39	5	53	C	67
(	40	6	54	D	68
)	41	7	55	E	69
*	42	8	56	F	70
+	43	9	57	G	71

Table of Display Characters and Corresponding ASCII Codes - continued on next page

Character	ASCII	Character	ASCII	Character	ASCII
H	72	b	98	|	124
I	73	c	99	}	125
J	74	d	100	→	126
K	75	e	101	←	127
L	76	f	102	α	224
M	77	g	103	ä	225
N	78	h	104	β	226
O	79	i	105	ε	227
P	80	j	106	μ	228
Q	81	k	107	σ	229
R	82	l	108	ρ	230
S	83	m	109	¢	236
T	84	n	110	£	237
U	85	o	111	ñ	238
V	86	p	112	ö	239
W	87	q	113	θ	242
X	88	r	114	∞	243
Y	89	s	115	Ω	244
Z	90	t	116	ü	245
[	91	u	117	Σ	246
¥	92	v	118	π	247
]	93	w	119	X	248
^	94	x	120	÷	253
_	95	y	121	▉	255
`	96	z	122		
a	97	{	123		

Table of Display Characters and Corresponding ASCII Codes

(Note: ASCII codes 0–7 provide access to eight user-definable characters (1–8). ASCII codes 16–32, 128–160, and 254 are blank spaces. ASCII codes 161–223 and 250–252 are Japanese characters. For a complete list of the display characters, contact the Videx Technical Support Department.)

CLOSE Statement

• **Action**

Closes the numbered file.

• **Syntax**

CLOSE [[#] *filenumber%* { **,** [#] *filenumber%* }]

• **Parameters**

Argument	Description
filenumber%	The file number used in the **OPEN** statement.

• **Remarks**

A **CLOSE** statement with no arguments closes all open files. All files can also be closed with an **END** statement.

See also **OPEN**, **SEEK**, and **PRINT**.

• **Example**

```
REM CLOSE Statement
DIM myFile$, i%
CLS
myFile$ = "data.txt"
OPEN myFile$ FOR APPEND as #0
FOR i% = 1 to 10         'write 10 lines with two tab
                         'delimited fields to the file
    PRINT #0, "LINE "; STR$(i%), "VIDEX DURATRAX"
NEXT i%
CLOSE #0
PRINT "Saved data to:"
PRINT myFile$;
SLEEP 0
```

CLS Statement

• **Action**

Clears the display.

• **Syntax**

CLS

• **Parameters**

None

• **Remarks**

After clearing the display, the cursor is placed in the top left corner (location **0,0**). It is not necessary to begin your program with a **CLS** statement, as the screen clears and the cursor is set to **0,0** when the program is executed.

See also **LOCATE**.

• **Example**

```
REM CLS Statement
NL$ = CHR$(13) + CHR$(10)
CLS
PRINT "CLS"; NL$; "Clears screen";
SLEEP 0
CLS
SLEEP 0
PRINT "See!"
SLEEP 0
```

Output:
```
CLS
Clears screen
See!
```

COMMCLOSE Statement

• **Action**

Closes the currently open serial port. (Videx BASIC)

• **Syntax**

COMMCLOSE

• **Parameters**

None

• **Remarks**

In the current operating system release, the serial port is always open at
9600 baud. The **COMMOPEN** and the **COMMCLOSE** statements are
ignored by the current operating system.

See also **COMMOPEN**, **COMMPRINT**, and **COMMINPUT**.

• **Example**

```
REM COMMCLOSE Statement

DIM someData$
someData$ = "This is some data"
CLS
PRINT "Press scan button"
PRINT "when ready.";
SLEEP 0
COMMOPEN 1,96,8,N,1
COMMPRINT someData$
COMMCLOSE
CLS
PRINT "Sent data out"
PRINT "the serial port!";
SLEEP 0
```

See also the **COMMINPUT** example.

COMMINPUT Statement

• Action

Gets a string from the currently open serial port and places the result in the named *stringvariable$*. (Videx BASIC)

• Syntax

COMMINPUT *stringvariable$* [**,** *timeout1%* [**,** *timeout2%*]]

• Parameters

Argument	Description
stringvariable$	This is where the string is placed that was taken from the currently open serial port.
timeout1%	The length of time to wait in milliseconds (ms) after the previous character is received before returning.
timeout2%	The length of time to wait (ms) for the first character before returning.

• Remarks

COMMINPUT returns when either *stringvariable$* is full or a timeout is reached.

The first timeout (*timeout1%*) is an intercharacter timeout (in ms). If a character is not received in the specified number of ms after the previous character was received, then **COMMINPUT** concludes that there are no more characters of input and returns immediately. The default for this value is zero.

The second timeout (*timeout2%*) is a first-character timeout (in ms). If the first character is not received in the specified number of ms after **COMMINPUT** is executed, then **COMMINPUT** concludes that there are no characters of input and returns immediately. The default for this value is the intercharacter timeout (*timeout1%*).

In most cases an array element can be used anywhere a variable can. **COMMINPUT** is one exception. The first argument cannot be an array element.

See also **COMMCLOSE**, **COMMOPEN**, and **COMMPRINT**.

• **Example**

```
REM COMMINPUT Statement

DIM accumData$ * 40,nowData$
CLS
PRINT "Press scan button"
PRINT "when ready.";
SLEEP 0
CLS
PRINT "    TERMINAL    "
PRINT "   EMULATION    ";
COMMOPEN 1,96,8,N,1
COMMPRINT "Waiting for you"
SLEEP 0
CLS
accumData$ = ""
DO
    nowData$ = ""
    COMMINPUT nowData$,5,10
    IF nowData$ <> "" THEN
        IF ASC(nowData$) = 27 THEN
            EXIT DO
        ELSE
            PRINT nowData$;
            accumData$ = accumData$ + nowData$
            IF LEN(accumData$) = 16 THEN
                LOCATE 1,0
            ELSEIF LEN(accumData$) > 32 THEN
                CLS
                accumData$ = ""
            ENDIF
        ENDIF
    ENDIF
LOOP
COMMCLOSE
CLS
```

COMMOPEN Statement

• **Action**

Opens a serial port with the specified settings. (Videx BASIC)

• **Syntax**

COMMOPEN *port%*, *baud%*, *databits%*, *parity$*, *stopbit%*

• **Parameters**

Argument	Description
port%	The serial port to open (use 1 for port 1).
baud%	Sets the baud rate (use 96 for 9600 baud).
databits%	Sets the databits (use 8 for 8 databits).
parity$	Sets the parity (use N for none).
stopbit%	Sets the stop bit (use 1).

• **Remarks**

Errors can be tested with the **ERR** function.

In the current operating system release, the serial port is always open at 9600 baud. The **COMMOPEN** and **COMMCLOSE** statements are ignored by the current operating system.

See also **COMMINPUT**, **COMMPRINT**, and **COMMCLOSE**.

• **Example**

See the **COMMINPUT** example.

All of the arguments must be exactly as shown in the example (**COMMOPEN 1, 96, 8, N, 1**). That is, port 1, baud 9600, 8 data bits, no parity, and 1 stop bit. The arguments are actually ignored, but the recommended arguments should be used to allow for future expansion.

COMMPRINT Statement

• **Action**

Sends characters out the serial port.

• **Syntax**

The term *expression* includes both *expression%* and *expression$*.

COMMPRINT [*expression*] { (**,** | **;**) [*expression*] }

• **Parameters**

Argument	Description
expression	The characters (integer (*expression%*) or string (*expression$*)) to be printed.

• **Remarks**

The **COMMPRINT** statement normally sends a carriage-return/line-feed pair at the end of the line. You can suppress this behavior by ending the **COMMPRINT** statement with a comma or semicolon. A comma sends a tab character. A semicolon separates arguments without sending anything out the port.

An error will occur if the statement contains two expressions in a row without a separator (however, it is acceptable to have two separators in a row).

See also **COMMINPUT**, **COMMOPEN**, and **COMMCLOSE**.

• **Example**

See the **COMMINPUT** example.

CONST Statement

• **Action**

Introduces a named constant value.

• **Syntax**

CONST *constantname* = *expression* {**,** *constantname* = *expression*}

• **Parameters**

Argument	Description
constantname	A name following the same rules as a BASIC variable name. You may add a type-declaration character (**%** or **$**) to the name to indicate its type, but the character is not part of the name.
expression	Any expression that can be reduced to a simple constant value; e.g., chr$(13) + "Hi!"

• **Remarks**

The constant name being introduced must not have been referred to on any previous lines.

If you use a type-declaration suffix in the name, you may omit the suffix character when the name is used, as shown in the following example:

```
CONST MAXDIM% = 250
.
.
.
DIM AccountNames$ (MAXDIM)
```

If you omit the type-declaration suffix, the constant is assumed to be an integer.

Constants must be defined before they are referenced. The following example produces an error because the constant **ONE** is not defined before it is used to define **TWO** (constants are defined from left to right):

```
CONST TWO = ONE + ONE, ONE = 1
```

A common programming practice is to use a statement like the following:

```
FALSE = 0
TRUE = NOT FALSE
```

Constants offer several advantages over using variables for constant values:

1. Constants can be defined only once for an entire module.
2. Constants cannot be inadvertently changed.
3. Constants produce more efficient code than using variables.
4. Constants make programs easier to modify.

The following program fragment declares a single constant to dimension a series of arrays. To increase or decrease the size of the arrays, it is necessary to change only the value of the **CONST** statement.

```
CONST MAXCUST = 250
   .
   .
   .
DIM AccountNumber$(MAXCUST), Balance(MAXCUST)
DIM Contact$(MAXCUST), PastDueAmount(MAXCUST)
   .
   .
   .
```

• Examples

```
REM CONST Statement
'* This example uses the NOT operator in a CONST expression.
CONST FALSE = 0, TRUE = NOT FALSE
CLS
PRINT "FALSE = "; STR$(FALSE)
PRINT "TRUE = "; STR$(TRUE);
BEEP
SLEEP 0
CLS
```

Output:

```
FALSE = 0
TRUE = -1
```

See also the **WHILE...WEND** example.

DATE$ Function

• Action

Returns a string containing the current date.

• Syntax

nowDate$ = DATE$ ( )

• Parameters

None

• Remarks

The **DATE$** function returns a ten-character string in the form *MM-DD-YYYY,* where *MM* is the month (01–12), *DD* is the day (01–31), and *YYYY* is the year (1964–2063).

See also **TIME$**.

• Returns

The date as a ten-character string.

• Example

```
REM DATE$() Function

Dim nowDate$, convDate$
CLS
nowDate$ = DATE$()
PRINT "Date is :"
PRINT nowDate$;
SLEEP 0
GOSUB make_date
CLS
PRINT "Date is :"
PRINT convDate$;
SLEEP 0
END

'
' make_date
' description: Convert an OS date "MM-DD-YYYY"
' into a TimeWand I style time "YYYYMMDD"
'

make_date:
    nowDate$ = DATE$
    convDate$ = RIGHT$(nowDate$, 4)
    convDate$ = convDate$ + LEFT$(nowDate$, 2)
    convDate$ = convDate$ + MID$(nowDate$, 4, 2)
RETURN
```

See also **TIME$** example number 1.

DIM Statement

• Action

A declaration statement that names one or more variables and allocates storage space for them.

• Syntax

DIM *variable$* [(*upperbound%*)] [* *max%*] {, single}

single

• Parameters

Argument	Description
variable$	A BASIC variable name.
upperbound%	The highest subscript (as an integer) to define the dimension of the array. The lowest subscript is always 0.
max%	The maximum size for a particular string.

• Remarks

In some cases, this is no different than just using the named variable in an expression. The name being introduced must not have been referred to on any previous line.

The syntax illustrates that more than one variable can be handled in a **DIM** statement. The term single represents allocating space for one variable; that is, *variable$* [(*upperbound%*)] [**max%*] = single.

If the *upperbound%* for an array is specified after the variable name, then an array is allocated with indexes from 0 to the *upperbound%*, inclusive. (Unlike some implementations of BASIC, it is only possible to have 0 for the lower bound of the array.)

If *max%* is specified for a string variable (after the * character), then the variable will be allocated with room for the specified number of characters. Otherwise, 31 characters is used by default.

The following statements are examples:

```
DIM A (5)    'integer array with 6 elements (0 to 5)
DIM B$*8     'string with maximum of 8 characters
DIM C$(5)*8 'an array of 6 strings, each with a
            'maximum 8 characters
DIM A (5), B$*8, C$(5)*8
            'the three previous examples on one
            'command line
```

It is good programming practice to put the required **DIM** statements at the beginning of the program outside of any loops.

• Example

```
REM DIM Statement

DIM question$(9) * 12     'dim an array of 10 questions;
                         'up to 12 chars each
FOR i = 0 to 9
   question$(i) = "QUESTION " + STR$(i+1)
                    'stuff array beginning with question 1
NEXT i

FOR i = 0 TO 9
   CLS
   PRINT "Element "; STR$(i) 'display the array's elements
   PRINT question$(i);
   SLEEP 2
NEXT i
```

DO...LOOP Statement

• **Action**

Repeats a block of statements (repeated while a condition is true or until a condition becomes true).

• **Syntax 1**

DO [(**WHILE** | **UNTIL**) *integerexpression%*]

[*statementblock*]

LOOP

• **Syntax 2**

DO

[*statementblock*]

LOOP [(**WHILE** | **UNTIL**) *integerexpression%*]

• **Parameters**

Argument	Description
integerexpression%	Any expression that evaluates to true (nonzero) or false (0).
statementblock	One or more BASIC statements to be repeated.

• **Remarks**

The **DO** condition is tested before each iteration of the loop, and the **LOOP** condition is tested after each iteration of the loop.

The *integerexpression%* is considered to be true if it is nonzero and false if it is zero.

You can use both conditions in the same loop, but typically one (and only one) condition will be used for any given loop. It is possible to use no conditions. In this case, the loop must be exited with an **EXIT** statement (or some other mechanism).

You may use a **DO...LOOP** statement instead of a **WHILE...WEND** statement. The **DO...LOOP** is more versatile because it can test for a condition at the beginning or at the end of a loop.

See also **WHILE...WEND** and **FOR...NEXT**.

• **Example**

The following two examples show how placement of the condition affects the number of times the block of statements is executed.

In the first example, the test is done at the beginning of the loop. Because **Q** is not less than 10, the body of the loop (the statement block) is never executed.

```
REM DO...LOOP Statement example 1

'DO...LOOP with test at top of loop.
'Output shows that loop was not executed.
DIM Q
Q = 10
PRINT "Beg. Q value:";Q
DO WHILE Q < 10
    Q = Q + 1
LOOP
PRINT "End Q value:";Q;
SLEEP 0
CLS
```

Output:

```
Beg. Q value: 10
End Q value: 10
```

The following example tests Q at the end of the loop, so the statement block executes at least once.

```
REM DO...LOOP Statement example 2

'DO...LOOP with test at bottom of loop.
'Output shows that loop was executed once.
DIM Q
Q = 10
PRINT "Beg. Q value:";Q
DO
    Q= Q + 1
LOOP WHILE Q < 10
PRINT "End Q value:";Q;

SLEEP 0
CLS
```

Output:

```
Beg. Q value: 10
End Q value: 11
```

In general, test at the end of a loop only if you know that you always want the body of the loop executed at least once.

END Statement

• **Action**

Terminates program.

• **Syntax**

END

• **Parameters**

None

• **Remarks**

By itself, the **END** statement stops program execution and closes all open files. The data collector returns to the command line, where it is possible to communicate with it over the serial port.

You may place **END** statements anywhere in the program to end program execution. The compiler always assumes an **END** statement at the conclusion of any program, so omitting an **END** statement at the end of a program still produces proper program termination.

• **Example**

```
REM END Statement

DIM buttonNum%
CLS
    PRINT "PRESS A"
    PRINT "BUTTON";
DO
    buttonNum% = ASC(INKEY$())
    IF buttonNum% <> 0 THEN
        BEEP
        END
    ENDIF
LOOP
```

ENVIRON$ Function

• **Action**

Returns various types of status information about the hardware and
environment. .

• **Syntax**

result$ = ENVIRON$ (*index%* **)**

• **Parameters**

Argument	Description
index%	An integer that determines the type of information returned. (0 returns battery voltage, 1 returns available RAM, 2 returns system version, 3 returns unit's ID)

• **Remarks**

The *index%* argument determines the type of information returned. See
the following table for examples:

Index	Status	Examples
0	Battery voltage	3.65
1	RAM available	500 24K
2	System version	1.0.0 DuraTrax OS
3	Identifier	0000000000

The available RAM is returned as bytes up to 1023, higher numbers are
returned as K.

• **Returns**

Returns a string variable with battery voltage, available RAM, system
version, or ID depending on *index%* used.

• Example

```
REM ENVIRON$() Function

DIM result$, envir%
result$ = ""
FOR envir% = 0 TO 3
    result$ = ENVIRON$(envir%)
    CLS
    IF envir% = 0 THEN
        PRINT "BATT VOLTS:"
    ELSEIF envir% = 1 THEN
        PRINT "AVAIL RAM:"
    ELSEIF envir% = 2 THEN
        PRINT "SYS VERSION:"
    ELSEIF envir% = 3 THEN
        PRINT "IDENTIFIER:"
    ENDIF
PRINT result$;
SLEEP 0
NEXT envir%
CLS
```

See also **LOF** function example 1.

EOF Function

• Action

Tests for the end-of-file condition.

• Syntax

result% = EOF (*filenumber%***)**

• Parameters

Argument	Description
filenumber%	The file number used in the **OPEN** statement (0–31).

• Remarks

The **EOF** function is used to test for the end-of-file condition of a file in RAM. See the **CARDCMD** statement to work with files on the LaserLite Mx memory card.

See also **OPEN**, **LOF**, and **LOFH**.

• Returns

Returns a **-1** if the end of a sequential file has been reached.

Note: This actually just returns true if the current file position is at the end of the file (after the last byte, ready to append). So, for example, **EOF(0)** would return true immediately after executing the statement:

```
SEEK #0, 0, E
```

- **Example**

```
REM EOF() Function
CONST false = 0,true = NOT false
DIM myFile$,i%,data$
myFile$ = "data.txt"
OPEN myFile$ FOR APPEND as #0     'this is an output file
FOR i% = 1 to 10                 'write 10 lines to the file
    PRINT #0, "LINE "; STR$(i%); "VIDEX DURATRAX"
NEXT i%
CLOSE #0
PRINT "Saved data to:"
PRINT myFile$;
SLEEP 0
CLS
dataLine$ = ""
OPEN myFile$ FOR APPEND as #0    'this is an input file
SEEK #0,0
DO WHILE NOT EOF(0)          'when EOF is reached, loop stops
    data$ = INPUT$(21,0)
    PRINT data$
    SLEEP 1         'pause screen for 1 second or keypress
LOOP
```

ERR Function

• **Action**

Returns the most recent error condition.

• **Syntax**

isError% = ERR

• **Parameters**

None

• **Remarks**

Statements that set the error condition are explicitly noted in this documentation. It is recommended that the **ERR** function be called after any file handling, serial communication, or LaserLite Mx memory card routines that set the error condition.

ERR clears the error condition.

• **Returns**

0 if no error; -1 if error is encountered.

• **Example**

See the **LOF** function example 2.

EXIT Statement

• **Action**

A control statement that exits a **FOR...NEXT** loop, **WHILE...WEND** loop, or a **DO...LOOP**. (Note: The **EXIT** statement functions somewhat differently in Videx BASIC than in QuickBASIC.)

• **Syntax**

EXIT [**FOR** | **WHILE** | **DO**]

• **Parameters**

None

• **Remarks**

If no loop is specified, exits the innermost enclosing loop of any kind. If a loop type is specified, exits the innermost enclosing loop of the specified kind.

• **EXIT FOR**
   An **EXIT FOR** statement may appear only in a **FOR...NEXT** loop. **EXIT FOR** transfers control to the statement following the **NEXT** statement.

   An **EXIT FOR** statement transfers out of the immediately enclosing loop when **FOR...NEXT** loops are nested.

• **EXIT WHILE**
   The **EXIT WHILE** statement can be used only inside a **WHILE...WEND** loop. **EXIT WHILE** transfers control to the statement following the **WEND** statement.

   An **EXIT WHILE** statement transfers out of the immediately enclosing loop when **WHILE...WEND** loops are nested.

- **EXIT DO**

  The **EXIT DO** statement can be used only inside a **DO...LOOP** statement. **EXIT DO** transfers control to the statement following the **LOOP** statement.

  An **EXIT DO** statement transfers out of the immediately enclosing loop when **DO...LOOP** statements are nested.

See also **FOR...NEXT**, **WHILE...WEND**, and **DO...LOOP**.

- **Example**

```
REM EXIT Statement (using EXIT DO)
DIM r%, k%, aKey$ * 4
r% = 1
k% = 0
aKey$ = ""

DO
    r% = r% + 1
    k% = ASC(INKEY$())
    LOCATE 0,0
    PRINT "Loop times " ; r%
    PRINT "ESC = Any key";
    IF r% > 1000 THEN
        CLS
        PRINT "Exited on..."
        PRINT "Loop = " ; r%;
        EXIT DO
    ELSEIF k% THEN
        IF k% = 1 THEN
            aKey$ = "SCAN"
        ELSEIF k% = 2 THEN
            aKey$ = "UP"
        ELSEIF k% = 4 THEN
            aKey$ = "DOWN"
        ENDIF
        CLS
        PRINT "You pressed the"
        PRINT aKey$;" Button";
        EXIT DO
    ENDIF
LOOP
SLEEP 0
```

FOR...NEXT Statement

• **Action**

Allows a series of instructions to be performed in a loop a given number of times.

• **Syntax**

FOR *counter%* = *start%* **TO** *end%* [**STEP** *increment%*]

.

.

.

NEXT [*counter%*]

• **Parameters**

Argument	Description
counter%	An integer variable used as the loop counter. The variable cannot be a record array.
start%	The initial value of the counter as an integer.
end%	The final value of the counter as an integer.
increment%	The amount the counter is incremented each time through the loop as an integer. This <u>must</u> be a constant expression.

• **Remarks**

The program lines following the **FOR** statement are executed until the **NEXT** statement is encountered. Then the loop counter (*counter%*) is adjusted by the amount specified by **STEP** (*increment%*), and compared with the final value (*end%*). (If you do not specify **STEP**, the increment is assumed to be one.) If the loop counter (*counter%*) is still not greater than the final value (*end%*), the control branches back to the statement after the **FOR** statement and the process is completed. When the loop counter (*counter%*) is greater than the final value (*end%*), execution continues with the statement following the **NEXT** statement.

If **STEP** (*increment%*) is negative, the loop counter (*counter%*) is
decreased each time through the loop, and the loop executes until the
counter is less than the final value (*end%*).

If the starting value (*start%*) is greater than the ending value (*end%*), the
loop does not execute at all. The following loop executes zero times:

```
FOR I=3 TO 2
    PRINT I
NEXT I
```

Do not change the value of a loop variable within the loop. Changing the
value can make the program more difficult to read and debug.

Assignment to the counter variable within the loop will have
unpredictable results.

If the name of a variable is used for the **TO** or **STEP** variable
expressions, then assignment to that variable within the loop will have
unpredictable results.

The following code fragment illustrates the problems:

```
FOR count% = 1 TO ending% STEP stepping%
    count% = 10                REM This is bad!
    ending% = 10               REM This is bad!
    stepping% = 10             REM This is bad!
NEXT count%
```

See also **DO...LOOP** and **WHILE...WEND**.

• **Example**

```
REM FOR...NEXT Statement
DIM i%
PRINT "    Euro    "
PRINT "   Sound   ";
FOR i% = 1 TO 4
   SOUND 4800, 500  '(frequency range 50-8000)
   SOUND 3300, 500  '(duration range 1-2000(2 seconds))
NEXT i%
```

Nested Loops

You can nest **FOR...NEXT** loops; in other words, you can place a **FOR...NEXT** loop within 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 the **NEXT** statement for the outside loop. The following construction is the correct form:

```
FOR I = 1 TO 10
    FOR J = 1 TO 10
        FOR K = 1 TO 10
          .
          .
          .
        NEXT K
    NEXT J
NEXT I
```

The **EXIT FOR** provides a convenient alternative exit to **FOR...NEXT** loops. See the **EXIT** statement for more information.

GOSUB...RETURN Statement

• **Action**

Branches to, and returns from a subroutine.

• **Syntax**

GOSUB *label*
.
.
.
RETURN

• **Parameters**

Argument	Description
label	The line label that is the first line of the subroutine.

• **Remarks**

Care must be taken not to execute a **RETURN** statement without first executing a **GOSUB** statement.

The program returns to the statement after the **GOSUB** statement.

You may call a subroutine any number of times in a program. You may also call a subroutine from within another subroutine. How deeply you can nest subroutines is limited only by the available stack space.

A subroutine may contain more than one **RETURN** statement. A **RETURN** statement in a subroutine makes BASIC branch back to the statement following the most recently executed **GOSUB** statement.

Subroutines may appear anywhere in the program, but it is good programming practice to make them readily distinguishable from the main program. To prevent inadvertent entry into a subroutine, precede it with an **END** or **GOTO** statement that directs program control around the subroutine.

• Example 1

```
REM GOSUB...RETURN Statement example 1

PRINT "main code";
SLEEP 0
GOSUB Sub1
Label1:
   CLS
   PRINT "back in main code";
   SLEEP 0
   END
Sub1:
   CLS
   PRINT "in sub 1";
   SLEEP 0
   GOSUB Sub2
Label2:
   CLS
   PRINT "back in sub 1";
   SLEEP 0
   RETURN
Sub2:
   CLS
   PRINT "in sub 2";
   SLEEP 0
   RETURN
```

Output:

```
main code
in sub 1
in sub 2
back in sub 1
back in main code
```

• Example 2

```
REM GOSUB...RETURN Statement example 2
DIM m%
CONST M1$ = "UP for Help"
CONST M2$ = "DOWN to Quit"
PRINT M1$
PRINT M2$;
DO
    GOSUB getMenu
LOOP
END

getMenu:
m% = 0
WHILE m% = 0
    m% = ASC(INKEY$())
WEND
CLS
IF m% = 1 THEN         rem scan button
    PRINT "   Wrong   "
    PRINT "   Button!   ";
    BEEP
    BEEP
ELSEIF m% = 2 THEN      rem up button
    PRINT "Help";
ELSEIF m% = 4 THEN      rem down button
    PRINT "Press Again"
    PRINT "to clear screen";
    SLEEP 5
    END
ENDIF
SLEEP 3
CLS
PRINT M1$
PRINT M2$;
RETURN
```

GOTO Statement

• **Action**

Branches unconditionally to the specified line.

• **Syntax**

GOTO *label$*

• **Parameters**

Argument	Description
label$	Line label to branch to.

• **Remarks**

The **GOTO** statement provides a way to branch unconditionally to another line (*label$*).

It is good programming practice to use structured control statements (**DO...LOOP**, **FOR...NEXT**, **IF...THEN...ELSE**) instead of **GOTO** statements because a program with many **GOTO** statements is difficult to read and debug.

• Example

```
rem GOTO Statement: This is a good example of how a control
rem structure, DO...LOOP, could be used instead of a GOTO.

DIM s%, k%, a%
k% = 0
s% = 0
a% = 0
PRINT "Press SCAN."
Start:
DO
    k% = ASC(INKEY$())
    IF k% > 0 THEN EXIT DO
LOOP
IF k% = 1 THEN
    BEEP
    END
ELSEIF k% = 2 THEN     rem up arrow increases
    s% = s% + 5
ELSEIF k% = 4 THEN     rem down arrow decreases
    IF s% > 5 THEN
    s% = s% - 5
  ENDIF
ENDIF

a% = s% * s%                  rem This example calculates
                              rem the area of a square.
CLS
PRINT "Sides = ";s%
PRINT "Area = "; a%;
SLEEP 0

GOTO Start
```

HEX$ Function

• **Action**

Produces a string of hexadecimal digits to represent a one-byte or two-byte integer value. (Videx BASIC) (Note: This function is not supported by the Macintosh version of the Videx BASIC compiler.)

• **Syntax**

result$ = HEX$ (*value%, num_digits%* **)**

• **Parameters**

Argument	Description
value%	The integer value to be converted to hexadecimal.
num_digits%	Number of characters to convert (0 to 4). If zero is specified, only the number of digits necessary to represent the value without leading zeros will be converted.

• **Remarks**

This function works similarly to the **CHR$** function but produces a string of hexadecimal digits (containing twice as many characters) that can represent binary numbers.

The characters will always be counted from the least-significant nibble (4 bits) of *value%*. The more significant characters are leftmost in the string produced. For example, if *value%* is equal to **2748** (0ABC hex) the resulting strings for all the values of *num_digits%* are:

num_digits%	result$
0	ABC
1	C
2	BC
3	ABC
4	0ABC

The compiler flags any out-of-range values for *num_digits%*. If the parameter is supplied by a variable, any out-of-range values are forced into range by the formula:

```
good  =  ( (bad - 1) MOD 4 ) + 1
```

For example, a value of 5 becomes 1 (see Example 1) and a value of 19 becomes 3 (see Example 2).

- Example 1
    good = ((**5** - 1) MOD 4) + 1
    good = (4 MOD 4) + 1
    good = (0) + 1
    good = 1

- Example 2
    good = ((**19** - 1) MOD 4) + 1
    good = (18 MOD 4) + 1
    good = (2) + 1
    good = 3

*Note: **MOD** arithmetic provides the remainder of an integer division, rather than the quotient. For example: **16 MOD 5 = 1**: $16 \div 5 = 3$ with a remainder **1**; **3250 MOD 256 = 178**: $3250 \div 256 = 12$ with a remainder of **178**. See page 18 for complete information on **MOD** arithmetic.*

Strings in Videx BASIC are null-terminated; that is, the character with the value zero has the special function of designating the end of the string. Because of that special designation, there is no way to include a character with the value zero within the string. In working with some types of data, for example, information stored in Touch Memory buttons, it is sometimes desirable to be able to handle strings of bytes that may contain binary information. The occurrence of bytes with a zero value is likely, so trying to use normal Videx BASIC strings for manipulating binary data would cause the data to be truncated at the first occurrence of a zero byte. Converting binary values to pairs of hexadecimal digits provides a way of transmitting binary information within a string.

See also **BIN** and **TOUCH**.

• **Returns**

Returns a string that is the hexadecimal representation of *value%* with a
specified number of characters.

• **Example**

```
REM BIN()and HEX$ Functions

DIM hex_data$ * 4
DIM i%, number%

FOR i% = 1 to 10

   number% = i% * 50
   hex_data$ = HEX$ (number%, 0)     'convert to hex
   CLS
   PRINT str$ (number%); " in hex:"
   PRINT hex_data$;                  'display result
   SLEEP 0
   number% = bin (hex_data$, 0)      'convert back to decimal
   CLS
   PRINT hex_data$; " as decimal:"
   PRINT str$ (number%);             'display result
   SLEEP 0

NEXT i%
END
```

IF...ELSEIF...ELSE...ENDIF Statement

• **Action**

Allows conditional execution and branching.

• **Syntax 1 (single line)**

IF *Booleanexpression%* **THEN** *thenpart*

• **Syntax 2 (block)**

IF *Booleanexpression1%* **THEN**
    [*statementblock-1*]
[**ELSEIF** *Booleanexpression2%* **THEN**
    [*statementblock-2*]]
.
.
.
[**ELSE**
    [*statementblock-n*]]
ENDIF

• **Parameters 1 (single line)**

The following list describes the parts of the single-line form:

Argument	Description
Booleanexpression%	Any expression that evaluates to true (nonzero) or false (zero).
thenpart	The statements or branches performed when *Booleanexpression%* is true (*thenpart*). The syntax is described below.

The *thenpart* has the following syntax:

{ *statements* | **GOTO** *linelabel* }

The following list describes the parts of the *thenpart* syntax:

Argument	Description
statements	One or more BASIC statements, separated by colons (:).
linelabel	A valid BASIC line label.

Note that **GOTO** is required with a line label.

Unlike other BASIC implementations, there is no **ELSE** clause, and only one statement may follow **THEN**.

• **Parameters 2 (block)**

The following list describes the parts of the block form:

Argument	Description
Booleanexpression1, Booleanexpression2	Any expression that evaluates to true (nonzero) or false (zero).
statementblock-1, statementblock-2, statementblock-n	One or more BASIC statements on one or more lines.

BASIC executes a block form **IF** by testing the first Boolean expression (*Booleanexpression1*). If the Boolean expression is true (nonzero), the statements following **THEN** are executed. If the first Boolean expression is false (zero), each **ELSEIF** condition is evaluated in turn. When a true condition is found, the statements following the associated **THEN** are executed. If none of the **ELSEIF** conditions are true, the statements following the **ELSE** are executed. After the statements following a **THEN** or **ELSE** are executed, the program continues with the statement following the **ENDIF**.

The **ELSE** and **ELSEIF** blocks are both optional. You can have as many **ELSEIF** clauses as you want in a block **IF**.

Any of the statement blocks can contain nested block **IF** statements.

BASIC looks at what appears after the **THEN** keyword to determine whether or not an **IF** statement is a block **IF**. If anything other than a

comment appears after **THEN**, the statement is treated as a single-line **IF** statement.

The block must end with an **ENDIF** statement.

ENDIF and **ELSEIF** statements may contain an optional space; for example: **END IF**, **ELSE IF**.

• **Remarks**

An **IF** statement can have any number of **ELSEIF** clauses, but only one **ELSE** clause. Both the **ELSEIF** and **ELSE** clauses are optional.

The single-line form of the statement is best used for short, straightforward tests where only one action is taken.

The block form provides several advantages:

- It provides more structure and flexibility than the single-line form by allowing conditional branches across several lines.

- It can test for more complex conditions.

- It lets you use the **THEN...ELSE** portion of the statement.

- It allows your program's structure to be guided by logic rather than by how many statements fit on a line.

Programs that use block-form **IF...THEN...ELSE** are usually easier to read, maintain, and debug.

The single-line form is never required. Any program using single-line **IF...THEN** statements can be written using the block form.

Note: The syntax of the single-line form is somewhat different in Videx BASIC than in QuickBASIC.

• **Example**

See the **GOTO** example.

INKEY$ Function

• **Action**

Retrieves and returns a character from the keyboard buffer.

• **Syntax**

thekey$ = INKEY$( )

• **Parameters**

None

• **Remarks**

The **INKEY$** function returns a one-byte string containing a character read from the input device, or a null string if no character is waiting. The one-character string contains the actual character read from the keyboard. The standard input device is the keyboard. **INKEY$** does not echo characters to the screen. (Note: Key combinations are not supported.)

See also **ASC** and **VAL**.

• **Returns**

A one-byte string containing a character read from the input device or a null string if no character is waiting. The one-character string contains the actual character read from the keyboard.

The following two tables list the keys and their corresponding ASCII values:

DuraTrax/LaserLite Keys

Key	ASCII Value
Scan button	1
Scroll up	2
Scroll down	4

LaserLite Pro/LaserLite Mx Keys

Key	ASCII Value	Key	ASCII Value
SHIFT (Unshifted)	128		
SHIFT (Shifted)	129	A	65
Scan button	1	B	66
ESC	27	C	67
ENTER	13 (Carriage return)	D	68
Backspace	8	E	69
0	48	F	70
1	49	G	71
2	50	H	72
3	51	I	73
4	52	J	74
5	53	K	75
6	54	L	76
7	55	M	77
8	56	N	78
9	57	O	79
-	45	P	80
.	46	Q	81
+	43	R	82
*	42	S	83
/	47	T	84
Scroll left	130	U	85
Scroll right	131	V	86
Scroll up	2	W	87
Scroll down	4	X	88
F1	134	Y	89
F2	135	Z	90
F3	136	Space	32
F4	137	MEM	132
F5	138	BAT	133

• **Example**

See the **GOTO** example.

INPUT$ Function

• **Action**

Returns a string of characters read from a specified file.

• **Syntax**

data$ = INPUT$ (*n%*, *filenumber%* **)**

• **Parameters**

Argument	Description
n%	The number of bytes to return; this is also the number of bytes the current file position is advanced.
filenumber%	The file number used in the **OPEN** statement.

• **Remarks**

Reads *n%* bytes from the sequential (data) file.

The file is always read directly into some string. If the string's maximum size is less than the number of requested bytes, then the input is truncated to the maximum size of the string. The current file position is always advanced by the number of bytes read, even if the input is truncated.

If **INPUT$** is used directly in an assignment statement, then the data is read directly into the assigned string. For example, in the following code sample, a string is allocated with a maximum size of 48 bytes. Then, 48 bytes are read from file 0, and the file position is advanced by 512 bytes.

```
DIM data$ * 48
data$ = INPUT$ (512, 0)
```

If **INPUT$** is used in an expression, then the data is read into a
temporary string with maximum size of 128. For example, in the
following code sample, a string is allocated with a maximum size of 512
bytes. Then, 128 bytes are read into a temporary string, and the file
position is advanced by 512 bytes. Finally, "suffix" is appended to the
temporary string and assigned to **data$**.

```
DIM data * 512
data$ = INPUT$ (512,0) + "suffix"
```

In general, it is best to allocate a string that is the same size as the
number of bytes you want to read, and then fill that string with an
assignment statement.

```
DIM data$ * 512
data$ = INPUT$ (512,0)
```

If you need to use an **INPUT$** function in an expression, be sure to read
fewer than 128 bytes, so the input isn't truncated.

See also **OPEN**.

The **INPUT$** function is used to read from a file in RAM. See the
CARDCMD statement to work with files on the LaserLite Mx memory
card.

• **Returns**

The character string read from the specified file.

• **Example**

See the **EOF** example.

INPUTEVT Statement

• **Action**

Waits for an event from the data collector and returns information about the event. (Videx BASIC)

• **Syntax**

INPUTEVT *left%, top%, right%, bottom%, TYPE%, SYMBOLOGY%, DEVICE%, DATA$*

• **Parameters**

Argument	Description
left%	An integer expression that represents the left side of the screen rectangle where data entry occurs.
top%	An integer expression that represents the top of the screen rectangle where data entry occurs.
right%	An integer expression that represents the right side of the screen rectangle where data entry occurs.
bottom%	An integer expression that represents the bottom of the screen rectangle where data entry occurs.
TYPE%	Variable that returns the type of event.
SYMBOLOGY%	Variable that returns the bar code's symbology.
DEVICE%	Variable that returns how the data was captured.
DATA$	String that contains the entered data (max 64 characters).

• **Remarks**

The four parameters (*left%, top%, right%, and bottom%*) define an area on the display for keypad or scanpad data entries. The rectangle **0, 0, 16, 2** is 16 characters wide, 2 characters tall, and starts in the upper-left corner. Rows and columns are numbered starting at zero, and *right%* and *bottom%* can be thought of as the first column and row that are NOT in the input area. If all four parameters are set to zero, it is still possible to enter data using the laser, contact bar code scanner, or Touch Memory button reader. If all four parameters are set to zero and data is entered using the keypad or scanpad, the data is accepted as single characters.

While waiting for an event, if nothing happens for 30 seconds, **INPUTEVT** automatically puts the data collector to sleep. It then sleeps

for 25 seconds and wakes for 5 seconds, without turning on the screen, this sleep cycle continues until an event occurs.

In most cases, an array element can be used anywhere a variable can. **INPUTEVT** is an exception; the last four arguments cannot be array elements.

• **Returns**

The parameters *TYPE%*, *SYMBOLOGY%*, *DEVICE%*, and *DATA$* must be variables. Values are assigned depending on the event that occurs.

TYPE% can be one of:

TYPE%	**Event**
1	Input
2	Exit
3	Delete last input (5 space character bar code scanned)
4	Scroll up key pressed
5	Scroll down key pressed
6	Power off
7	Unexpected power loss detected (occurs if the lock switch is not turned off before replacing the batteries)
8	LaserLite Pro or LaserLite Mx ESC key (escape) pressed
9	Scan key pressed and released
10	LaserLite Mx no memory card module found
11	LaserLite Mx no memory card found
12	LaserLite Mx unknown memory card found
13	LaserLite Mx memory card ID has changed
14	LaserLite Mx memory card processor interrupted
18	LaserLite Pro or LaserLite Mx scroll left key pressed
19	LaserLite Pro or LaserLite Mx scroll right key pressed
20	LaserLite Pro or LaserLite Mx MEM key (memory) pressed
21	LaserLite Pro or LaserLite Mx BAT key (battery) pressed
22	LaserLite Pro or LaserLite Mx f1 key pressed
23	LaserLite Pro or LaserLite Mx f2 key pressed
24	LaserLite Pro or LaserLite Mx f3 key pressed
25	LaserLite Pro or LaserLite Mx f4 key pressed
26	LaserLite Pro or LaserLite Mx f5 key pressed

LaserLite Mx uses all of the *TYPE%* events; DuraTrax and LaserLite use events 1–7 and 9; and the LaserLite Pro uses 1–9 and 18–26.

Note: Event 7 (unexpected power loss detected) is not always detected by the operating system. If event 7 occurs, the application starts running from the beginning and sometimes returns with event 7 the first time it executes **INPUTEVT**. Any data not yet written to the data file is lost.

DEVICE% can be one of:

1	keyboard
2	laser
12	contact scanner
48	button touch

SYMBOLOGY% can be one of:

1	Code 39
2	UPC-A or UPC-E
4	Interleaved 2 of 5
8	Codabar
16	EAN
64	UCC Code 128 (starts with reserved character Fn1)
128	Code 128 (standard: no Fn1 character at start)

DATA$ is a string containing the data which is entered. The maximum number of characters in this string is 64; longer input strings are not supported by the operating system.

If *TYPE%* is 1, then the user has entered data. *DEVICE%* gets the device that was used to enter it, *SYMBOLOGY%* gets the symbology of the entered data, and *DATA$* gets the entered data.

• Example

```
REM INPUTEVT Statement
'*
'* Reports the origin of data.
'*
  CONST NL$ = CHR$(13) + CHR$(10)
'carriage return/line feed combination
  running% = 1
  display$ = "Waiting for  " + NL$ + "you....   "
                                 'preprompt user
WHILE running%                   'event loop
    CLS
    PRINT display$;              'display a prompt
    INPUTEVT 0, 0, 0, 0, type%, symbol%, device%, data$
                                 'get event
    ON type% GOSUB fn_null, fn_input, fn_exit, fn_null, fn_up, fn_dn, fn_power
WEND
END
fn_null:               'description:  Do nothing function.

RETURN

'*
'*   fn_input
'*      description: Respond to a user input event. Beeps and
'*      blinks the Valid Scan LED, and then appends the input
'*      to end of the data file, along with current date and
'*      time.
'*

fn_input:

  OPTION( 258) = 1                          'turn on the LED
  SOUND 2933, 250                           'good beep
  OPTION( 258) = 0                          'turn off the LED
  GOSUB get_origin
  display$ = origin$ + NL$ + data$  'display input on screen
RETURN

fn_exit:            'Respond to an exit event
  running% = 0      'Exit events are generated when an unlock
RETURN              'command is sent to the unit during an
                    'INPUTEVT statement

fn_up:              'Respond to a scroll-up event, notify
  SOUND 3000, 225   'user that we got the keypress
RETURN

fn_dn:              'Respond to a scroll-down event, notify
  SOUND 1000, 225   'user that we got the keypress
RETURN
```

```
'*
'*    fn_power
'*        description: Respond to a power event. These events
'*        are generated when the power switch is turned off.
'*        Respond by putting the data collector into a low-power
'*        state.
'*

fn_power:

  OPTION(256) = 0  'turn off the LCD (not needed for Mx)
  SOUND 4300, 500
  SOUND 3300, 500
  SOUND 2300, 500
  SOUND 1300, 500
  SLEEP 0                'sleep indefinitely
  OPTION(256) = 1  'turn LCD back on (not needed for Mx)

RETURN

get_origin:

  IF device% = 1 THEN
    origin$ = "keyboard"
  ELSEIF device% and 14 THEN        'laser or contact scanner
    IF symbol% = 1 THEN
      origin$ = "USS-39"
    ELSEIF symbol% = 2 THEN
      origin$ = "UPC"
    ELSEIF symbol% = 4 THEN
      origin$ = "USS-I 2/5"
    ELSEIF symbol% = 8 THEN
      origin$ = "Codabar"
    ELSEIF symbol% = 16 THEN
      origin$ = "EAN"
    ELSEIF symbol% = 64 THEN
      origin$ = "UCC Code 128"
    ELSEIF symbol% = 128 THEN
      origin$ = "USS 128"
    ELSE
      origin$ = "unknown"
    ENDIF
  ELSEIF device% and 48 THEN        'touch button
    origin$ = LEFT$(data$, 2) 'get the family code, add 30
    origin$ = "button type " + STR$(30 + VAL(origin$))
    data$ = RIGHT$(data$, 12) 'convert data into ROM code
  ELSE
    origin$ = "unknown"
  ENDIF
RETURN
```

INSTR Function

• **Action**

Returns the character position of the first occurrence of a string in another string.

• **Syntax**

position% = INSTR ([*start%*,] *stringexpression1$*, *stringexpression2$* **)**

• **Parameters**

Argument	Description
start%	An optional offset that sets the position for starting the search; *start%* must be an integer in the range 1–32767. If *start%* is not given, the **INSTR** function begins the search at the first character of *stringexpression1$*.
stringexpression1$	The string being searched.
stringexpression2$	The string to search for.

The arguments *stringexpression1$* and *stringexpression2$* can be string variables, string expressions, or string literals.

See also **LEN**, **MID$**, **LEFT$**, and **RIGHT$**.

• **Remarks**

Searches for an occurrence of *stringexpression2$* in *stringexpression1$*.

Use the **LEN** function to find the length of *stringexpression1$*.

• **Returns**

The value returned by **INSTR** depends on the following conditions:

Condition	Returned Value
stringexpression2$* found in *stringexpression1$	The position where the match is found.
start% greater than length of *stringexpression1$*	0
stringexpression1$ is null string	0
stringexpression2$ cannot be found	0
stringexpression2$ is null string	*start%* (if given); otherwise 1

• **Example**

```
REM INSTR() Function

data$ = "two words"
CLS
PRINT "Splitting..."
PRINT data$;
SLEEP 3
blankPosition% = INSTR(1,data$," ")
firstWord$ = LEFT$(data$,blankPosition%-1)
secondword$ = RIGHT$(data$,LEN(data$) - blankPosition%)
CLS
PRINT "Word 1 is ";firstWord$
PRINT "Word 2 is ";secondWord$;
SLEEP 3
END
```

Output:

```
Splitting...
two words
Word 1 is two
Word 2 is words
```

LCASE$ Function

• **Action**

Returns a string with all letters in lowercase.

• **Syntax**

word$ = LCASE$ (*stringexpression$* **)**

• **Parameters**

Argument	Description
stringexpression$	The string to lowercase. This can be a string variable, string constant, or string expression.

• **Remarks**

LCASE$ is helpful in string comparison operations where tests need to be case insensitive.

See also **UCASE$**.

• **Returns**

The string in lowercase letters.

• **Example**

```
REM LCASE$() Function

word$ = "LOWERCase"
PRINT word$
word$ = LCASE$(word$)
PRINT word$;
SLEEP 0
```

Output:

```
LOWERCase
lowercase
```

LEFT$ Function

• **Action**

Returns a string that is composed of the leftmost characters in the string argument.

• **Syntax**

word$ = LEFT$ (*stringexpression$*, *n%* **)**

• **Parameters**

Argument	Description
stringexpression$	Any string variable, string constant, or string expression.
n%	An integer expression (range 0–32767) indicating how many characters are to be returned.

• **Remarks**

See also **INSTR**, **LEN**, **MID$**, and **RIGHT$**.

• **Returns**

If *n%* is greater than the number of characters in *stringexpression$*, the entire string is returned. To find the number of characters in *stringexpression$*, use the **LEN** function.

If *n%* is zero, a null string (length zero) is returned.

• Example 1

```
REM LEFT$() Function example 1

word$ = "BASIC LANGUAGE"
PRINT word$
B$ = LEFT$(word$,5)
'get first 5 characters (should be BASIC)
PRINT B$;
SLEEP 0        'pause display until keypress
B$ = RIGHT$(word$,8)
'get last 8 characters (should be LANGUAGE)
LOCATE 1,0
PRINT B$;
SLEEP 0        'pause display until keypress
```

Output:
```
BASIC LANGUAGE
BASIC
LANGUAGE
```

• Example 2

```
REM LEFT$() Function example 2

data$ = "two words"
CLS
PRINT "Splitting..."
PRINT data$;
SLEEP 3                'pause display for 3 seconds or keypress
blankPosition% = INSTR(1,data$," ")
firstWord$ = LEFT$(data$,blankPosition%-1)
secondword$ = RIGHT$(data$,LEN(data$) - blankPosition%)
CLS
PRINT "Word 1 is ";firstWord$
PRINT "Word 2 is ";secondWord$;
SLEEP 3
END
```

Output:
```
Splitting...
two words
Word 1 is two
Word 2 is words
```

See also **INSTR** example.

LEN Function

• **Action**

Returns the number of characters in a string.

• **Syntax**

length% = LEN (*stringexpression$* **)**

• **Parameters**

Argument	Description
stringexpression$	Any string variable, string constant, or string expression.

• **Remarks**

See also **INSTR**, **LEFT$**, **MID$**, and **RIGHT$**.

• **Returns**

The number of characters in the argument *stringexpression$*.

• **Example**

```
REM LEN() Function

LET word$ = "Videx DuraTrax"
PRINT word$
lenWord% = LEN(word$)
length$ = STR$(lenWord%)
PRINT "is "; length$; " chars.";
SLEEP 0        'pause display until keypress
```

Output:
```
Videx DuraTrax
is 14 chars.
```

LET Statement

• **Action**

Assigns the value of an expression to the named variable.

• **Syntax 1**

 [**LET**] *variable$* = *stringexpression$*

• **Syntax 2**

 [**LET**] *variable%* = *integerexpression%*

• **Parameters**

Argument	Description
variable$	The name of a string variable where the result of the string expression will be copied.
variable%	The name of an integer variable to receive the value of the integer expression.
stringexpression$	A string constant, variable, or single value obtained by combining constants, variables, and other expressions with operators.
integerexpression%	An integer constant, variable, or single value obtained by combining constants, variables, and other expressions with operators.

• Remarks

The keyword **LET** is optional. The equal sign in the statement is enough to inform BASIC that the statement is an assignment statement. However, it is somewhat faster to compile code that uses the **LET** statement and the compiled program is identical. Both *stringvariable$* and *integervariable%* can be an element of an array.

The corresponding lines, in the following two program fragments, perform the same functions:

```
LET D=12                        D=12
LET E=12-2                      E=12-2
LET F=12-4                      F=12-4
LET SUM=D+E+F                   SUM=D+E+F
.                               .
.                               .
.                               .
```

• Example

```
REM LET() Statement

LET newLine$ = CHR$(13) + CHR$(10)   'carriage return/
                                     'line feed combination
PRINT "Line 1"; newLine$; "Line 2";
SLEEP 0                       'pause display until keypress
```

Output:

```
Line 1
Line 2
```

LOCATE Statement

• Action

Moves the cursor to the specified row and column.

• Syntax

LOCATE *row%, column%*

• Parameters

Argument	Description
row%	The number of a row on the screen; *row%* is an integer number.
column%	The number of a column on the screen; *column%* is an integer number.

• Remarks

The rows and columns are numbered from 0. The first character of the next **PRINT** statement will be at the designated location.

• Example 1

```
LOCATE 0,0    'Moves cursor to the upper-left corner (1st
              'character position) of the first line on the
              'display.
LOCATE 1,5    'Moves cursor to the 6th character position of
              'the second line on the display.
```

• Example 2

```
REM LOCATE Statement
PRINT "This line fixed"          'displayed on line 1
PRINT "This line changes...";    'displayed on line 2
SLEEP 0
FOR i% = 1 TO 10
   LOCATE 1,0        'position cursor at start of line 2
   PRINT "New Line ";i%;"     ";         'print on line 2
   SLEEP 1           'a 1 second delay for reading
NEXT i%
SLEEP 0              'press key to exit
```

LOF Function

• **Action**

Returns the length of the file in bytes. If the length of the file is greater than 32767, only the low-order word of the length is returned.

• **Syntax**

fileLen% = LOF (*filenumber%* **)**

• **Parameters**

Argument	Description
filenumber%	The number of the file used in the **OPEN** statement.

• **Remarks**

See also **EOF**, **LOFH**, **OPEN**, and **SEEK**.

• **Returns**

For files opened as reference, returns the total number of records in the file.

For sequential (data) files, returns the size of the file **MOD 32768**.

The **LOF** function is used to check the length of a file in RAM. See the **CARDCMD** statement to work with files on the LaserLite Mx memory card.

The following two examples each have a program line that is longer than the width of the page; so, we will use the pipe (|) character as an indication that the program line continues.

• Example 1

```
REM LOF() Function    example 1
'* This example creates a new file with 70,000 bytes,
'* works around the short integer limit, and prints
'* file size as two integers (an integer that is a
'* multiple of 32678, and an integer of the remaining
'* bytes before the next multiple integer).
'* It also reports the data collector's remaining memory.
myFile$ = "data.txt"
multipleLen% = 0
byteLen% = 0
PRINT "Building file"
PRINT "Please wait";
OPEN myFile$ FOR APPEND AS #0     'this is output file
FOR i% = 1 TO 700
'write 700 lines with 100 bytes to the file
    PRINT #0,"01234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890"
NEXT i%
CLOSE #0
CLS
PRINT "Saved data to:"
PRINT myFile$;
SLEEP 0
OPEN myFile$ FOR APPEND AS #0     'this is input file
multipleLen% = LOFH (0)    'this is a multiple of 32768
byteLen% = LOF (0)         'this is the number of bytes
                           'before the next multiple
CLS
PRINT "Len = ";multipleLen ; " X 32,768"
PRINT "+ " ; byteLen;
SLEEP 0
CLS
PRINT "Memory is"
PRINT ENVIRON$(1);
SLEEP 0
CLOSE #0
```

• Example 2

```
REM LOF() Function example 2
'* This example illustrates the use of LOF to determine
'* the number of records in a file opened for reference.
'* Note: Requires the Lof2.crf to be loaded with the
'* example.

refFile$ = "LOF2.CRF"
nl$ = CHR$(13) + CHR$(10)
OPEN refFile$ FOR REFERENCE as #1
   IF ERR THEN
      PRINT "Can't open file"; nl$; refFile$;
      SLEEP 0
      END
   ENDIF
   recLen% = LOF (1)  'this is number of records in file
   PRINT refFile$;" has..."; nl$; recLen%; "total records.";
SLEEP 0
CLOSE #1
END
```

LOF Statement

• **Action**

Sets the size of the file.

• **Syntax**

LOF (*filenumber%* **) =** [*intexpr2%* **,**] *intexpr3%*

• **Parameters**

Argument	Description
filenumber%	The file number used in the **OPEN** statement.
intexpr2%	Whole number multiple of 32768.
intexpr3%	Remainder over whole number multiple.

• **Remarks**

The **LOF** statement is useful for clearing or truncating a file.

For sequential (data) files, the **LOF** statement sets the size of the file. If the first argument isn't used, it is assumed to be 0. The total size of the file is set to ***intexpr2% * 32768 + intexpr3%***.

Note: If either argument is negative, the results are undefined.

The **LOF** statement is used to set the size of a file in RAM. See the **CARDCMD** statement to work with files on the LaserLite Mx memory card.

• Example

```
REM LOF() Statement
myFile$ = "data.txt"
PRINT "Building file."
PRINT "Please wait.";
OPEN myFile$ FOR APPEND AS #0     'this is output file
FOR i% = 1 TO 3000
'write 3000 lines with 10 bytes to the file
    PRINT #0, "012345678901234567890"
NEXT i%
CLOSE #0
CLS
PRINT "Saved data to:"
PRINT myFile$;
SLEEP 0
OPEN myFile$ FOR APPEND AS #0     'this is input file
byteLen% = LOF (0)                'get length of file
CLS
PRINT "length is  " ; byteLen
PRINT "purging last 1/2";
SLEEP 0
LOF (0) = byteLen%/2              'set file length to 1/2
CLS
byteLen% = LOF (0)         'get length of file now
PRINT "length is  " ; byteLen
PRINT "Clearing it.";
SLEEP 0
LOF (0) = 0         'set file length to zero and clears
                    'contents
CLOSE #0
```

LOFH Function

• **Action**

Returns the high-order word of the number of bytes in the numbered file.

• **Syntax**

multipleLen% = LOFH (*filenumber%* **)**

• **Parameter**

Argument	Description
filenumber%	The file number used in the **OPEN** statement.

• **Remarks**

For sequential (data) files, returns **the size of the file / 32768**. Since an integer can only represent numbers in the range -32768 to 32767, and files can be much larger than that, the size of a file must be represented with two integers. The true size of the file is:

```
LOFH () * 32768 + LOF ()
```

See also **EOF**, **LOF** function, **OPEN**, and **SEEK**.

The **LOFH** function is used to check the length of a file in RAM. See the **CARDCMD** statement to work with files on the LaserLite Mx memory card.

• **Returns**

The size of the file as an integer.

• **Example**

See the **LOF** function example number 1.

LOOK$ Function

• **Action**

Returns the data from a designated field in a cross-reference file. The structure of a cross-reference file is composed of four fields: Input (key field), Field1, Field2, and Field3. (Videx BASIC)

• **Syntax**

fieldDesc$ = LOOK$ (*fieldnum%* **)**

• **Parameters**

Argument	Description
fieldnum%	The field value to be returned. If *fieldnum%* is 0 then the key field itself is returned.

• **Remarks**

LOOK$ and **LOOKUP** are used in a program to retrieve data from a cross-reference file. **LOOKUP** is used first, then **LOOK$**. **LOOKUP** designates which file number to look in and what data to look for. When **LOOKUP** finds the data, it returns the data's row number and sets a pointer to that row in the cross-reference file. **LOOK$** is then used to designate the column of the cross-reference file that contains the data.

For example, using the cross-reference file below:

	0	1	2	3
	Input (key field)	Field1	Field2	Field3
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5

The following program line:

```
rec% = LOOKUP (filenumber%, A3)
```

would search for **A3** in the file, which it finds in row 3; **rec%** is now equal to **3**. The next program line:

```
fieldDesc$ = LOOK$ (1)
```

would go to column 1 of the row designated in **LOOKUP** and return its value; in this case, it would return **B3**; **fieldDesc$** is now equal to **B3**.

If the user assumes the wrong number of fields in a cross-reference file, then the results are unpredictable.

Note: You can actually use cross-reference files with up to 16 fields by using **Vxcrf.exe**. See the *DuraTrax, LaserLite, & LaserLite Pro Developer's Reference Manual* for information on using **Vxcrf.exe**.

See also **LOOKUP** and **OPEN**.

The **LOOK$** function is used with cross-reference files in RAM. See the **CARDCMD** statement to work with cross-reference files on the LaserLite Mx memory card.

• **Returns**

If **LOOKUP** returns a nonzero value, then **LOOK$**(n) returns the nth value associated with the found key in the file. **LOOK$**(0) returns the key field itself. If **LOOKUP** returns 0, then the value of **LOOK$( )** is undefined.

• Example

```
REM LOOK$() Function and LOOKUP Function
'* This example looks up data$ in the LOF2.CRF file and if
'* matched, displays the data$ that was searched for, the
'* record number of the matching entry, and the field being
'* displayed (these 3 values are displayed on line 1). Line
'* 2 displays the actual description of each field.
'* NOTE: You may in turn use rec% to quickly find the
'* index record for editing by using the following syntax:
'* rec% = LOOKUP(1,rec%)
data$ = "5E"
OPEN "LOF2.CRF" FOR REFERENCE AS #1      'open cross-ref file
    IF ERR THEN
        GOSUB crf_error     'if missing, give an error
    ENDIF
rec% = LOOKUP(1, data$)    'look for the data$ (5E) in
                           'the input field of file and
                           'store its record number
IF rec% THEN               'found it
    BEEP                       'so let them know with a beep
    field1$ = LOOK$(1)     'get description from field1
    CLS
    PRINT data$;" #";rec%;" Fld 1"
    PRINT field1$;
    SLEEP 5
    field2$ = LOOK$(2)     'get description from field2
    CLS
    PRINT data$;" #";rec%;" Fld 2"
    PRINT field2$;
    SLEEP 5
    field3$ = LOOK$(3)     'get description from field3
    CLS
    PRINT data$;" #";rec%;" Fld 3"
    PRINT field3$;
    SLEEP 5
ENDIF
END

crf_error:
BEEP
CLS
PRINT "Could not open"
PRINT "LOF2.CRF";
SLEEP 2
RETURN
```

LOOKUP Function

• **Action**

Searches the specified cross-reference file for the specified key field and sets the **LOOK** pointer to that row in the file. (Videx BASIC)

• **Syntax**

rec% = LOOKUP (*filenumber%,* (*strexpr$ | intexpr%*)**)**

• **Parameters**

Argument	Description
filenumber%	The file number used in the **OPEN** statement.
strexpr$, intexpr%	The data string (*strexpr$*) or the record number (*intexpr%*) to look for.

• **Remarks**

The specified file must have been previously opened for reference, and must be properly formatted as a reference file. (A properly formatted cross-reference file is one that was created or opened with Application Builder or converted with **Vxcrf.exe** or **Vxcrfw.exe**. See the *DuraTrax, LaserLite, & LaserLite Pro Developer's Reference Manual* for information on **Vxcrf.exe** and **Vxcrfw.exe**.)

Each record in a reference file is randomly assigned a unique number from 1 through *n* (where *n* is the number of records in the file). This unique ID is returned from **LOOKUP**. If the second argument to **LOOKUP** is an integer instead of a string, it looks up the record with that unique ID (instead of searching for a key).

See also **LOOK$** and **OPEN**.

The **LOOKUP** function is used to work with cross-reference files in RAM. See the **CARDCMD** statement to work with cross-reference files on the LaserLite Mx memory card.

• Returns

If the key is found, its ID is returned; otherwise, 0 is returned.

• Example

See the **LOOK$** example.

LTRIM$ Function

• **Action**

Returns a copy of a string with leading spaces removed.

• **Syntax**

data$ = LTRIM$ (*stringexpression$* **)**

• **Parameters**

Argument	Description
stringexpression$	Any string expression.

• **Remarks**

See also **RTRIM$**.

• **Returns**

Copy of string with no leading spaces.

• Example

```
REM LTRIM$() Function and RTRIM$() Function
myFile$ = "data.txt"
LET NL$ = CHR$(13) + CHR$(10)
PRINT "Building file..."
PRINT "Please wait"
OPEN myFile$ FOR APPEND AS #0     rem this is output file
FOR i% = 1 TO 10                  rem write 10 lines to file
    data$ = "LINE" + STR$(i%) + "."
    PRINT #0, data$
NEXT i%
CLOSE #0
CLS
PRINT "Saved data to:"
PRINT myFile$;
SLEEP 0
dataLine$ = ""
OPEN myFile$ FOR APPEND AS #0     rem this is input file
SEEK #0,0
DO
    data$ = TOKEN$ (0,NL$,0)
    IF data$ = "" THEN EXIT DO
    CLS
    PRINT CHR$(34); data$; CHR$(34)
    PRINT CHR$(34); LTRIM$(RTRIM$(data$)); CHR$(34);
    SLEEP 1
LOOP
CLOSE #0
SLEEP 0
```

MID$ Function

• **Action**

Returns a substring of a string.

• **Syntax**

result$ = MID$ (*stringexpression$*, *start%* [, *length%*] **)**

• **Parameter**

Argument	Description
stringexpression$	The string that the substring is extracted from; this can be any string expression.
start%	Designates the character position in the string expression where the substring starts.
length%	Designates the number of characters to extract.

• **Remarks**

The *start%* and *length%* arguments must be in the range of 1–32767. If the *length%* argument is omitted or if there are fewer than *length%* characters to the right of the *start%* character, the **MID$** function returns all characters to the right of the *start%* character.

If *start%* is greater than the number of characters in the string expression, **MID$** returns a null string.

Use the **LEN** function to find the number of characters in a string expression.

See also **LEFT$**, **LEN**, **MID$** statement, and **RIGHT$**.

• **Returns**

All characters to the right of the *start%* character.

The following example has a program line that exceeds the width of the page; so, we will use the pipe (|) character to indicate that the program line continues.

• Example

```
REM MID$() Function
DIM toSend$ * 500
toSend$ = "Sending this sentence to the screen and out the|
serial port one character at a time."

PRINT "Press when..."
PRINT "ready.";
SLEEP 0
FOR i% = 1 TO LEN(toSend$)
   theChar$ = MID$(toSend$,i%,1)
   CLS
   PRINT "Sending..."; theChar$;
   COMMPRINT theChar$
   GOSUB slow_down
NEXT i%
CLS
PRINT "Finished job"
SLEEP 0
END

slow_down:          'this subroutine is just a delay to
FOR r = 1 TO 200    'display the data longer
NEXT r
RETURN
```

MID$ Statement

• Action

Replaces a portion of a string variable with another string.

• Syntax

MID$ (*stringvariable$***,** *start%* [**,** *length%*] **)** = *stringexpression$*

• Parameters

Argument	Description
stringvariable$	The string variable being modified.
start%	A numeric expression giving the position in *stringvariable$* where the replacement starts.
length%	The number of characters to replace. The length is a numeric expression.
stringexpression$	The string expression that replaces part of the string variable.

• Remarks

If *length%* is not specified, then substitution continues until the end of either *stringexpression$* or *stringvariable$* is reached.

See also **LEFT$**, **LEN**, **MID$** function, and **RIGHT$**.

• Example

```
REM MID$() Statement

LET test$ = "Paris, France"
LOCATE 0,0
PRINT test$
MID$(test$,8) = "Texas "   'replace the characters in test$,
LOCATE 1,0    'starting with the 8th char, with "Texas"
PRINT test$;
SLEEP 3       'pause display for 3 seconds or until keypress
```

Output:

```
Paris, France
Paris, Texas
```

ON…GOSUB, ON...GOTO Statement

• Action

Branches to one of a list of subroutines, depending on the value of an expression.

• Syntax 1

ON *expression$* **GOSUB** *label* { **,** *label* }

• Syntax 2

ON *expression$* **GOTO** *label* { **,** *label* }

• Parameters

Argument	Description
expression$	Determines which *label* the program branches to.
label	The subroutine to branch to.

• Remarks

The *expression$* argument can be any numeric expression. If the value is 0, then the first subroutine is executed. If the value is 1, then the second subroutine is executed, and so on.

The *label* arguments are a list of the subroutines separated by commas.

If the value of *expression$* is out of range, then the line is skipped, and the next line is executed.

The following example has a program line that exceeds the width of the page; so, we will use the pipe (|) character to indicate that the program line continues.

• Example

```
REM ON...GOSUB Statement
'*
'* Monitors events in the data collector and calls
'* subroutines.
'*
CONST false = 0, true = Not false, NL$ = CHR$(13) + CHR$(10)
DIM display$ * 34
running% = true                    'set running to true
display$ = "Waiting on you..." + NL$ + "P switch = exit"
                                   'preprompt them
WHILE running%                     'event loop
   CLS
   PRINT display$;                 'display a prompt of some kind
   INPUTEVT 0, 0, 0, 0, type%, symbol%, device%, data$
                                   'get an event
   ON type% GOSUB fn_null, fn_input, fn_exit, fn_null,|
   fn_up, fn_dn, fn_power
WEND

END

fn_null:              'does nothing

RETURN

fn_input:            'an input routine goes here
CLS
BEEP
PRINT "Input routine"
SLEEP 2
RETURN

fn_exit:            'an exit routine goes here
   running% = false%
RETURN

fn_up:              'a scroll up routine goes here
   SOUND 2933, 125     'notify user that we got the keypress
RETURN

fn_dn:              'a scroll down routine goes here
   SOUND 2933, 125     'notify user that we got the keypress
RETURN

fn_power:
   OPTION( 256) = 0    'turn off the LCD
   SLEEP 0             'sleep indefinitely
   OPTION( 256) = 1    'turn the LCD back on
RETURN
```

OPEN Statement

• **Action**

Opens named file.

• **Syntax**

OPEN *file$* **FOR** (**APPEND**|**REFERENCE**) **AS** [**#**] *filenumber%*

• **Parameters**

Argument	Description
file$	A string expression that specifies a filename.
filenumber%	An integer expression between 0 and 31. When a **OPEN** is executed, this number is associated with the file as long as it is open. Other I/O statements may use the number to refer to the file.

• **Remarks**

If the file is opened for **APPEND**, text data can be printed to the file. On the DuraTrax, LaserLite, LaserLite Pro, and LaserLite Mx it is only possible to open a single file in RAM for append (this is the data file).

The name most recently used to open the data file is the name by which it is saved when it is transferred to the computer.

If the file is opened for **REFERENCE**, the file is assumed to be properly formatted as a cross-reference file. (A properly formatted cross-reference file is one that was created or opened with Application Builder or converted with **Vxcrf.exe** or **Vxcrfw.exe**. See the *DuraTrax, LaserLite, & LaserLite Pro Developer's Reference Manual* for information on **Vxcrf.exe** and **Vxcrfw.exe**.) **LOOKUP** can be called for files opened for **REFERENCE**.

The file number must be between 0 and 31. File numbers greater than 31 result in an error.

It is recommended that the **ERR** function be called after the **OPEN** statement or other file handling routines.

See also **CLOSE**, **EOF**, **LOF**, **LOFH**, **LOOKUP**, and **SEEK**.

The **OPEN** statement is used to open a file in RAM. See the **CARDCMD** statement to open files on the LaserLite Mx memory card.

• **Example**

See the **LOF** function examples 1 and 2.

OPTION Function

• **Action**

Returns the status of an environment option . (Videx BASIC)

• **Syntax**

result% = OPTION (*optionnumber%* **)**

• **Parameters**

Argument	Description
optionnumber%	The option number.

• **Remarks**

Returns the value of options less than 255. The value of options less than 128 are returned as true (-1) or false (0).

See also **OPTION** statement.

• **Returns**

The status of the environment option.

OPTION Statement

• **Action**

Sets the value of an environment option.

• **Syntax**

OPTION (*optionnumber%* **) =** *integerexpression%*

• **Parameters**

Argument	Description
optionnumber%	The option number.
integerexpression%	The value to which the option should be set. For options less than 128, this is any non-zero value to enable the option and zero to disable the option. Options greater than 128 can be set to a value within the specified range.

• **Remarks**

Following is a list of available options:

Option	Number	Range	Default
Read Code 3 of 9	8	True 1-False 0	1 (True)

Enables reading Code 3 of 9 symbology bar codes. When Code 3 of 9 is
disabled, the data collector cannot decode Code 3 of 9 symbology bar codes.

Read Interleaved 2 of 5	9	True 1-False 0	1 (True)

Enables reading Interleaved 2 of 5 (I 2of5) symbology bar codes. When
Interleaved 2 of 5 is disabled, data collector cannot decode I 2of5 bar codes.

Read UPC/EAN	10	True 1-False 0	1 (True)

Enables reading UPC and EAN symbology bar codes. When UPC/EAN is
disabled, the data collector cannot decode UPC or EAN symbology bar codes.

Read Codabar	11	True 1-False 0	1 (True)

Enables reading Codabar symbology bar codes. When Codabar is disabled, the
data collector cannot decode Codabar symbology bar codes.

Option	Number	Range	Default

Read Code 128　　　　　　　　　12　　True 1-False 0　　1 (True)
Enables reading Code 128 symbology bar codes. When Code 128 is disabled, the data collector cannot decode Code 128 symbology bar codes (including UCC128).

Require valid checksum for Code 3 of 9
　　　　　　　　　　　　　　　　16　　True 1-False 0　　0 (False)
When this option is enabled, the data collector can only decode a Code 3 of 9 bar code if it has a valid modulo 43 symbol check character. The check character is determined as follows:

1. Assign a numerical value to each data character in the symbol as shown in the table below.
2. Sum the numerical values for all of the data characters and divide the sum by 43.
3. The remainder obtained in step 2 is the value of the check character shown in the table.

Character Values for Code 3 of 9 Modulo 43 Check Character Calculation

Character	Value	Character	Value	Character	Value
0	0	F	15	U	30
1	1	G	16	V	31
2	2	H	17	W	32
3	3	I	18	X	33
4	4	J	19	Y	34
5	5	K	20	Z	35
6	6	L	21	-	36
7	7	M	22	.	37
8	8	N	23	SPACE	38
9	9	O	24	$	39
A	10	P	25	/	40
B	11	Q	26	+	41
C	12	R	27	%	42
D	13	S	28		
E	14	T	29		

Example:

For the data message: CODE 39
C=12, O=24, D=13, E=14, SPACE=38, 3=3, 9=9
12 + 24 + 13 + 14 + 38 + 3 + 9 = 113
113/43 = 2 with a remainder of 27
27 = R
Resultant data with check character: CODE 39R

Option	Number	Range	Default

Transmit Code 3 of 9 checksum   17   True 1-False 0   0 (False)

If this option is enabled and option 16 is enabled, the data collector places both the bar code and the check character in the scan buffer. If this option is disabled but option 16 is enabled, the data collector strips the check character from the bar code data. For example:

- In the previous example, the bar code data with checksum is **CODE 39R**.
- The data collector returns **CODE 39R** with option 17 enabled.
- The data collector returns **CODE 39** with option 17 disabled.

Require valid checksum for Code I 2 of 5

                                        20   True 1-False 0   0 (False)

When this option is enabled, the data collector only decodes an Interleaved 2 of 5 bar code if it has a valid weighted modulo 10 symbol check character. The weighted check character is determined as follows:

1. Starting at either end of the string of data characters, multiply all of the odd position characters by 3.
2. Sum the products obtained in step 1, along with the remaining even position data characters, and divide the sum by 10.
3. If the remainder obtained in step 2 is 0, the value of the check digit is 0. Otherwise, subtract the remainder from 10. The result of this subtraction is the check digit.
4. Append the check digit to the end of the data.

   Example:

Data Digits: 4 3 8 2 7
Weights: 3 1 3 1 3
Weighted Sum = (3 x 4) + (1 x 3) + (3 x 8) + (1 x 2) + (3 x 7) = 62
62/10 = 6 with a remainder of 2
10 - 2 = 8
Therefore, the check digit is 8.
Data with check digit is: 4 3 8 2 7 8

Option	Number	Range	Default

Transmit Code I 2 of 5 checksum   21   True 1-False 0   0 (False)
If this option is enabled and option 20 is enabled, the data collector places both
the bar code and the check character in the scan buffer. If it is disabled, but
option 20 is enabled, the data collector strips the check character from the bar
code data.

For example:
- In the previous example, the bar code data with checksum is **438278**.
- The data collector returns **438278** with option 21 enabled.
- The data collector returns **43827** with option 21 disabled.

Expand UPC-E to UPC-A form   24   True 1-False 0   0 (False)
UPC-E is a shortened version of the UPC-A symbology where redundant zeros
are removed according to a specific formula. If this option is enabled, the data
collector expands the UPC-E bar code to its equivalent UPC-A form.

UPC-E bar code

UPC-A bar code showing same data
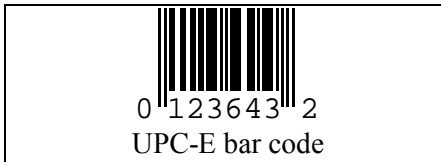
Transmit check character for UPC 25   True 1-False 0   1 (True)
The rightmost character in each of the bar code symbols above is the check
character for the bar code. Option 25 toggles whether the check character is to be
included with the bar code data when it is returned by **INPUTEVT**.

Option	Number	Range	Default

Transmit country code character for UPC-A

26   True 1-False 0   0 (False)

UPC symbols are a subset of the more comprehensive EAN system. EAN-13 symbols encode the first digit in the parity pattern of the characters on the left side of the symbol. For UPC codes all these characters have odd parity, which gives a zero in the EAN scheme. A UPC-A symbol is therefore equivalent to an EAN-13 symbol with a first digit of zero. This digit is called the country code, since its function (sometimes along with the second character) in the EAN system is to specify the country of origin.

Country Code transmission disabled:
UPC-A   012300000642
0 = number system code 1230000064 = data 2 = check character

In the bar code above, toggling the country code character would have the following results:

Country Code transmission enabled:
UPC-A   0012300000642
0 = country code 0 = number system code 1230000064 = data 2 = check character

The Uniform Code Council recommends disabling transmission of the country code character. To transmit the country code for a UPC-E bar code, it must first be expanded to the UPC-A form by enabling option 24.

Transmit number system character for UPC

27   True 1-False 0   1 (True)

The first digit of a 12-character UPC-A symbol represents the number system of the code; it is the digit usually printed to the left of the code. If this option is enabled, the data collector includes the number system character with the bar code data.

Option	Number	Range	Default

Report UPC-A source as EAN   28   True 1-False 0   0 (False)

A UPC-A symbol is actually an EAN-13 symbol with a country code of zero. If this option is enabled, the data collector reports the symbology code from a UPC bar code as 16. If this option is disabled, it reports the origin code as 2.

Transmit check character for EAN

                                  30   True 1-False 0   1 (True)

The rightmost character in each of the EAN bar code symbols below is the check character for the bar code. Option 30 toggles whether the check character is to be included with the bar code data.

EAN- 8 bar code
Option 30 disabled - 0034000
Option 30 enabled - 00340007

EAN-13 bar code
Option 30 disabled - 780045486123
Option 30 enabled - 7800454861232
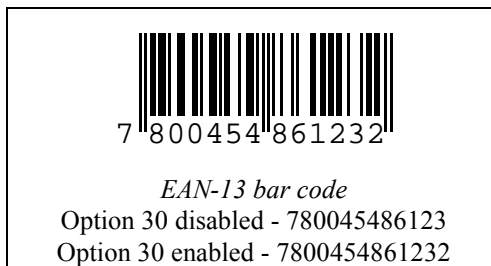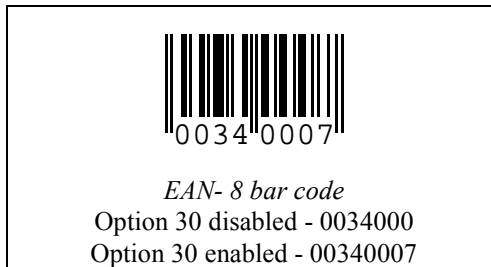
Option	**Number**	**Range**	**Default**

Allow supplement with UPC or EAN

31 True 1-False 0 1 (True)

The data collector decodes both 2-digit and 5-digit add-ons to UPC or EAN bar codes. Enabling this option, returns the supplement data with the scan data, if a supplement is detected. The supplement is separated from the main code by a space character. Options 32 and 33 must be disabled for this option to be effective.

UPC-A bar code with 2 character extension

EAN-13 bar code with 5 character extension
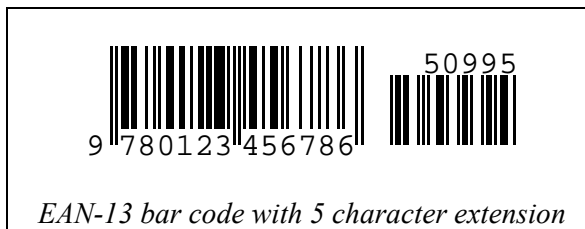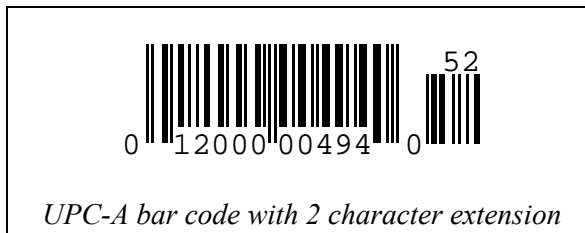
Require supplement for UPC or EAN

32 True 1-False 0 0 (False)

If this option is enabled <u>and</u> option 33 is disabled, the data collector cannot decode a UPC or EAN bar code without a supplement. Option 32, when enabled, takes precedence over option 31.

Ignore any UPC or EAN supplement

33 True 1-False 0 0 (False)

If this option is enabled, the data collector decodes the UPC or EAN symbol regardless of whether there is a supplement. Option 33, when enabled, takes precedence over options 31 and 32.

UPC/EAN Supplement Options

The following table details the interaction of the options that control the handling of supplements appended to UPC or EAN bar codes. For each option, the table entries represent the following states:

      T - Option is enabled or true.
      F - Option is disabled or false.
      X - The state of the option does not affect the result.

The entries for each supplement state describe the data returned when such a bar code is scanned and the options are in the configuration described in the same column.

Allow supplement (Option 31)	X	X	T	F
Require supplement (Option 32)	X	T	F	F
Ignore supplement (Option 33)	T	F	F	F
Bar code with good supplement	Returns Base only	Returns Base + Supplement	Returns Base + Supplement	Returns Error
Bar code with no supplement	Returns Base only	Returns Error	Returns Base only	Returns Base only
Bar code with bad supplement[†]	Returns Base only	Returns Error	Returns Error	Returns Error

[†] The system can see that a supplement is present but it does not decode correctly. It is possible to scan a bar code symbol in such a way that the system cannot detect the presence of the supplement.

Option	Number	Range	Default

Transmit Codabar start and stop characters

36   True 1-False 0   1 (True)

The Codabar symbology uses **A**, **B**, **C**, or **D** as a start or stop character. Enabling this option includes the start and stop characters in the scan data.

Enable inkspread correction   37   True 1-False 0   1 (True)

When bar codes are printed using toner or ink based printers, the toner or ink may "spread" beyond its intended bounds. This may cause the nominal width of the bars to exceed the nominal width of the spaces in the bar code. The data collector's decoding algorithm includes a correction for light to moderate instances of inkspread. This can help increase decode reliability and overall readability of certain bar codes. Enabling this option enables the inkspread correction routine.

Minimum quiet zone (ratio to smallest bar)

131   6–64   6

The quiet zone of a bar code is the clear space that precedes the start character of a bar code symbol and follows the stop character. The width of the quiet zone is expressed as a multiplier of the width of the narrow bars and narrow spaces (X-dimension). The minimum recommended quiet zone for a Code 3 of 9 symbology bar code is 10 X-dimension, that is 10 times the width of the narrow bars and narrow spaces of the bar code symbol. The space between the main symbol and the supplement of a UPC or EAN bar code must also be a quiet zone and its minimum width is 7 X-dimension. For the DuraTrax, LaserLite, LaserLite Pro, or LaserLite Mx, the minimum quiet zone for all instances is 6 X-dimension or 6 times the width of the narrow bars. This option allows setting the minimum quiet zone ratio in the range of 6 to 64. It can be set to 10 or 15, for example, to decode Code 3 of 9 bar codes that have large spaces between the characters, so that the spaces are not interpreted as the end of the bar code.

The following options set the minimum and maximum length of bar code for each symbology. The maximum size of a data element is limited to 64 characters by internal registers. It may also be further limited by the storage space allocated in the DESCRIBE.SRC template for any variables that must contain the data. (Note: See the *DuraTrax, LaserLite, & LaserLite Pro Developer's Reference Manual* for information on the DESCRIBE.SRC template used by the Application Builder software.)

Option	Number	Range	Default
Shortest Code 39 bar code accepted	136	1–64	1
Shortest I 2 of 5 bar code accepted	137	1–64	4
Shortest Codabar bar code accepted	139	1–64	1
Shortest Code 128 bar code accepted	140	1–64	1
Longest Code 39 bar code accepted	144	1–64	64
Longest I 2 of 5 bar code accepted	145	1–64	64
Longest Codabar bar code accepted	147	1–64	64
Longest Code 128 bar code accepted	148	1–64	64

Following are some additional options:

Option	Description
256	Screen on or off (write only).
258	LED 2 (Valid Scan LED) on or off (write only).
259	Key mode on or off (write only).
512...575	Sets the programmable display character lines.

The programmable display characters options (512–575) allow you to design up to 8 characters (ASCII 0 through 7) for use on the data collector's display. The character size is 5 x 8 pixels.

Use options 512–519 to create character 0.
Use options 520–527 to create character 1.
Use options 528–535 to create character 2.
Use options 536–543 to create character 3.
Use options 544–551 to create character 4.
Use options 552–559 to create character 5.
Use options 560–567 to create character 6.
Use options 568–575 to create character 7.
(Note: Application Builder uses character 7.)

In this example, we will create a character for character 1, using options 520–527. The first step is to design your character on a grid similar to the one shown in the following diagram. Once you design the character, you then define each option line used.

To define each option line, add the values (listed at the top) for the black pixels. For example, line 520 in the diagram below, has a black pixel at the 4 value; this line is defined as: **Option (520) = 4**. Line 521 has a black pixel at the 8 and the 2 values. Add the values together and this line is defined as: **Option (521) = 10**.

The character lines in this diagram would be defined as:

```
        16  8   4   2   1
520    [ ][ ][■][ ][ ]        Option (520) = 4
                             Option (521) = 10
521    [ ][■][ ][■][ ]        Option (522) = 10
                             Option (523) = 21
522    [ ][■][ ][■][ ]        Option (524) = 17
                             Option (525) = 10
523    [■][ ][■][ ][■]        Option (526) = 10
                             Option (527) = 4
524    [■][ ][ ][ ][■]

525    [ ][■][ ][■][ ]

526    [ ][■][ ][■][ ]

527    [ ][ ][■][ ][ ]
```

The programmable display character options do not take effect until the screen is turned off and back on, i.e.   **Option (256) = 0**
                                                             Option (256) = 1

• Example

```
REM OPTION Statement
'Repeats the good read sound and flashes the LED 6 times.
PRINT "GOOD READ"
PRINT "SEQUENCE";
FOR i = 1 TO 6
    OPTION( 258) = 1 'turn on the LED
    SOUND 1446, 250  'good beep
    OPTION( 258) = 0 'turn off the LED
    GOSUB killTime
NEXT i
END

killTime:     'just a short delay for demo purposes only
FOR r = 1 TO 200
NEXT r
RETURN
```

PATTERN Function

• **Action**

Determines whether the second string matches the TimeWand II-style pattern in the first string.

• **Syntax**

matchPat% = PATTERN (*stringexpression1$***,** *stringexpression2$* **)**

• **Parameters**

Argument	Description
stringexpression1$	String to match (TimeWand II-style pattern).
stringexpression2$	String being compared.

• **Remarks**

When specifying a TimeWand II-style pattern, five characters are used as wild cards to create the pattern. These characters are:

#	A number [0-9] must be in this position.
@	A letter [a..z, A..Z] must be in this position.
&	Any character can be in this position.
=	The rest of the string can be any characters or no character.
>	The rest of the string must be numbers [0..9] or no character.

These are the actual characters your bar codes may contain:

    A..Z   0..9   space   +   -   /   %   . $

Following are examples of TimeWand II-style patterns:

Pattern	Acceptable Entries
E=	Any entry that starts with the letter E.
E>	Any entry that starts with the letter E and is followed only by numeric digits.
E0003	Entry must match E0003 exactly.
#>	At least one digit followed by any number of digits.
AAA=	Any entry that begins with AAA.
AA#A	A four character long entry, with A in the first, second, and fourth positions; and a number in the third position.
AA@A	Same as previous entry, but with a letter in the third position.
AA&A	Same as previous entry, but with any character in the third position.

• Returns

If *stringexpression2$* matches the TimeWand II-style pattern in *stringexpression1$,* **PATTERN** returns a -1; otherwise, it returns 0.

• Example

```
REM PATTERN() Function    'checks two strings against pattern
data1$ = "555-AB-1234"    'great tool for restricting input
data2$ = "555-55-1234"
matchPat$ = "###-##-####"
PRINT matchPat$
PRINT data1$;
c% = PATTERN(matchPat$,data1$)
GOSUB show_result
PRINT matchPat$
PRINT data2$;
c% = PATTERN(matchPat$,data2$)
GOSUB show_result
END

show_result:
SLEEP 2   'pause display for 2 seconds or until keypress
CLS
PRINT "PATTERN IS.."
IF c% = 0 THEN
   PRINT "DIFFERENT";
ELSE
   PRINT "THE SAME";
ENDIF
SLEEP 0              'pause display until keypress
CLS
RETURN
```

PRINT Statement

• **Action**

Outputs data to the screen or to a file.

• **Syntax**

The term *expression* includes both *expression%* and *expression$*.

(**PRINT** | **?**) [**#** *filenumber%***,**] [*expression*] {(**,** | **;**) [expression]}

• **Parameters**

Arguments	Description
filenumber%	The file number used in the **OPEN** statement.
expression	The string or integer expression to be printed.

• **Remarks**

The **PRINT** statement normally prints a carriage-return/line-feed pair at the end of the line. You can suppress this behavior by ending the **PRINT** statement with a comma or semicolon. A comma moves the current pen location to the next tab stop. A semicolon leaves the pen at its current location. An error occurs if the statement contains two expressions in a row without a separator (however, it is acceptable to have two separators in a row).

If a file number is specified, the data is printed to the previously **OPEN**ed data file rather than to the screen. In that case, the error condition may be set. You should always check the error condition when writing to the data file to minimize data loss. The error condition should also be checked after any **PRINT** statement that uses a concatenation symbol (+) to combine strings.

See also **LOCATE** and **OPEN**.

The **PRINT** statement is used to print to a file in RAM. See the
CARDCMD statement to write to files on the LaserLite Mx memory
card.

The data collector's LCD can display the following characters:

!	"	#	$	%	&	'	(	)	*
+	,	-	.	/	@	:	;	<	>
=	?	{	}	|	0	1	2	3	4
5	6	7	8	9	A	a	ä	B	b
C	c	D	d	E	e	F	f	G	g
H	h	I	I	J	j	K	k	L	l
M	m	N	n	ñ	O	o	ö	P	p
Q	q	R	r	S	s	T	t	U	u
ü	V	v	W	w	X	x	Y	y	Z
z	Ω	μ	Σ	π	¢	°	£	←	→

• **Example**

```
REM PRINT Statement
'* Prints to a file and to the screen.
myFile$ = "data.txt"
PRINT "Now printing..."
PRINT "Please wait";
OPEN myFile$ FOR APPEND AS #0     'this is output file
FOR i% = 1 TO 1000  'send the data$ contents 1000 times
    LOCATE 1,0
    data$ = "Line number " + ;i
    PRINT data$;      'data$ goes to second line of display
    PRINT #0, data$          'data$ is appended to a file
NEXT i%
CLOSE #0
CLS
PRINT "Done"
SLEEP 3                      'pause display for 3 seconds
```

REM Statement

• **Action**

Allows explanatory remarks to be inserted in a program.

• **Syntax**

(**REM** | **'**) { *anything* }

• **Parameters**

Argument	Description
anything	This can be any combination of letters, numbers, symbols, etc.

• **Remarks**

The compiler ignores everything from **REM** to the end of the line. **REM** can appear at the end of any line, including a line that has other BASIC statements. You may branch from a **GOTO** or **GOSUB** statement to a **REM** statement. Execution continues with the first executable statement after the **REM** statement.

• **Example**

```
'REM Statement note that you may use REM or ' to remark a line
DIM myArray(23)
FOR I = 0 TO 22 : myArray(I) = 0 : NEXT I    REM Initialize array
FOR I = 0 TO 22 : myArray(I) = 0 : NEXT I     'Initialize array
```

RIGHT$ Function

• **Action**

Returns a string that is composed of the rightmost characters in the string argument.

• **Syntax**

data$ = RIGHT$ **(** *stringexpression$*, *n%* **)**

• **Parameters**

Argument	Description
stringexpression$	Any string variable, string constant, or string expression.
n%	A numeric expression (range 0–32767) indicating how many characters are to be returned.

• **Remarks**

See also **LEFT$**, **LEN**, and **MID$**.

• **Returns**

If *n%* is greater than the number of characters in *stringexpression$*, the entire string is returned. To find the number of characters in *stringexpression%*, use the **LEN** function.

If the integer argument is zero, a null string (length zero) is returned.

• **Example**

See the **LEFT$** example.

RTRIM$ Function

• **Action**

Returns a copy of a string with trailing (right-hand) spaces removed.

• **Syntax**

data$ = RTRIM$ (*stringexpression$* **)**

• **Parameters**

Argument	Description
stringexpression$	Any string expression.

• **Remarks**

See also **LTRIM$**.

• **Returns**

Copy of string with no trailing spaces.

• **Example**

See the **LTRIM$** example.

SEEK Function

• **Action**

Returns the current file position.

• **Syntax**

cfp% = SEEK (*filenumber%* **)**

• **Parameters**

Argument	Description
filenumber%	The file number used in the **OPEN** statement.

• **Remarks**

See also **EOF**, **OPEN**, **SEEK** statement, and **SEEKH**.

The **SEEK** function is used to work with a file in RAM. See the **CARDCMD** statement to work with files on the LaserLite Mx memory card.

• **Returns**

The byte position in the file where the next operation is to take place. The first byte in a file is 0.

For sequential (data) files, returns the current file position (from the beginning of the file) **MOD 32768**.

• Example

```
REM SEEK() Function
DIM data$ * 10
myFile$ = "data.txt"
OPEN myFile$ FOR APPEND AS #0     'this is output file
FOR i% = 1 TO 30              'write 30 lines to the file
    PRINT #0, "LINE "; STR$(i%); " VIDEX LASERLITE PRO"
NEXT i%
CLOSE #0
PRINT "Saved data to:"
PRINT myFile$;
SLEEP 0
CLS
dataLine$ = ""
OPEN myFile$ FOR APPEND AS #0     'this is input file
SEEK #0,0                     'go to the beginning of file
DO WHILE NOT EOF(0)          'when EOF is reached, loop stops
   data$ = INPUT$(50,0)    'input 50 bytes and stuff into
                           'the 10 byte data$ variable
   CLS
   PRINT data$                   'print the data$ variable
   cfp% = SEEK(0)               'return current file position
   PRINT "Position = ";cfp%;          'print it
   SLEEP 1          'pause display for 1 second or keypress
LOOP
CLS
PRINT "END OF FILE"
SLEEP 2              'pause display up to 2 seconds
```

SEEK Statement

• **Action**

Sets the position in a sequential (data) file for the next read or write.

• **Syntax**

SEEK [**#**] *filenumber%*, *position%* [**, (S | E | P)**]

• **Parameters**

Argument	Description
filenumber%	The file number used in the **OPEN** statement.
position%	An offset from the start, end, or current file position.
**S	E	P**	**S** = from start; **E** = from end; **P** = from current position.

• **Remarks**

Set the position in a sequential (data) file for the next read or write. The file position can be specified as an offset from the start of the file, the end of the file, or the current file position. Note that the file position can be set to an arbitrarily large number by calling **SEEK** repeatedly with an offset from the current position. Also note that **SEEK #0, 0** sets the file position at the first byte in the file, as opposed to other implementations of BASIC, where **SEEK #0, 1** does the same thing.

See also **EOF**, **OPEN**, **SEEK**, and **SEEKH**.

The **SEEK** statement is used to work with a file in RAM. See the **CARDCMD** statement to work with files on the LaserLite Mx memory card.

• **Example**

```
SEEK #0, 0       'set file position at first byte in file
SEEK #0, 0, S    'set file position at first byte in file
SEEK #0, 0, E    'set file position at first byte after file
                 '(prepare to append)
SEEK #0, 10, P   'advance the file position 10 bytes
SEEK #0, -10, P  'move the file position back 10 bytes
```

See the **EOF** and **SEEK** function examples.

SEEKH Function

• **Action**

Returns the high-order word of the current file position.

• **Syntax**

cHp% = SEEKH (*filenumber%* **)**

• **Parameters**

Argument	Description
filenumber%	The file number used in the **OPEN** statement.

• **Remarks**

Since an integer can only represent numbers in the range -32768 to 32767, and files can be much larger than that, the size of a file must be represented with two integers.

The true file position is:

```
SEEKH() * 32768 + SEEK()
```

See also **SEEK**.

The **SEEKH** function is used to work with a file in RAM. See the **CARDCMD** statement to work with files on the LaserLite Mx memory card.

• **Returns**

For sequential (data) files, returns the current file position (from the beginning of the file) / 32768.

• Example

```
REM SEEKH() Function
'* Also uses the SEEK() function.


DIM data$ * 10
myFile$ = "data.txt"
PRINT "Please wait..."
PRINT "making BIG file";
OPEN myFile$ FOR APPEND AS #0          'this is an output file
FOR i% = 1 TO 1500              'write 1500 x 50 (75K) to the file
PRINT #0,"12345678901234567890123456789012345678901234567890"
NEXT i%
CLOSE #0
CLS
PRINT "Saved data to:"
PRINT myFile$;
SLEEP 0
CLS
dataLine$ = ""
OPEN myFile$ FOR APPEND AS #0           'this is an input file
SEEK #0,0                               'go to beginning of file
DO WHILE NOT EOF(0)             'when EOF is reached, loop stops
    data$ = INPUT$(10000,0)    'input 50 bytes and stuff into the
                               '10 byte data$ variable

    CLS
    PRINT "Position is..."
    cHp% = SEEKH(0)            'return current file position
    cfp% = SEEK(0)
    msg$ = ""
    IF cHp%  > 0 THEN
        IF cHp% = 1 THEN
            msg$ = "32,767 + "
        ELSE
            msg$ = "32,767*" + STR$(cHp%) + " + "
        ENDIF
    ENDIF
    PRINT msg$;cfp%;    'print it
    SLEEP 0             'pause display for 1 second or keypress
LOOP
CLS
PRINT "END OF FILE"
SLEEP 2                 'pause display for up to 2 seconds
```

SGN Function

• **Action**

Indicates the sign of a numeric expression.

• **Syntax**

theSign% = SGN (*intexpression%* **)**

• **Parameters**

Argument	Description
intexpression%	Any integer expression.

• **Remarks**

None

• **Returns**

1 if integer is positive; 0 if integer is 0; -1 if integer is negative.

• Example

```
REM SGN() Function

DIM quantity(7)              'dim an array of 8 values

quantity(0) = 1
quantity(1) = -2
quantity(2) = 3
quantity(3) = -4
quantity(4) = 5
quantity(5) = 6
quantity(6) = -7
quantity(7) = 8

FOR i = 0 TO 7
    CLS
    IF SGN(quantity(i)) = 1 THEN
        theSign$ = "positive"
    ELSE
        theSign$ = "negative"
    ENDIF
    PRINT "The value..."
    PRINT quantity(i);" is "; theSign$;
    SLEEP 2
NEXT i
```

SLEEP Statement

• **Action**

Pauses program execution for a specified number of seconds. (Videx BASIC)

• **Syntax 1** (for DuraTrax, LaserLite, and LaserLite Pro)

SLEEP *seconds%*

• **Parameters 1**

Argument	Description
seconds%	Number of seconds the data collector stays asleep once it enters sleep mode.

• **Remarks 1**

If the number of seconds is 0, then the program execution is paused indefinitely. Any event will cause program execution to resume. A subsequent **INPUTEVT** statement does not report key events, but other events are reported.

The data collector is in a low-power state while it's sleeping. The IrDA chip is off, so serial communications are not received. The screen remains active for the duration of the **SLEEP** command, but it is possible to turn the screen off before going to sleep to enter the lowest possible power (see the **OPTION** statement).

When debugging, use the **SLEEP** statement to pause program execution while displaying intermediate results on the screen.

• **Example 1**

```
SLEEP 0              'waits for a keypress
SLEEP 3              'waits for 3 seconds or a keypress
```

See the **PATTERN** example.

• **Syntax 2** (for LaserLite Mx)

SLEEP *param%*

• **Parameters 2**

Argument	Description
param%	Length of time the LaserLite Mx's LCD is controlled by the **SLEEP** statement.

• **Remarks 2**

For the LaserLite Mx, the value of *param%* specifies the LCD state when the system is asleep as follows:

param% = 0	LCD is off; system sleeps indefinitely.
param% < 0	LCD is off; system sleeps for the number of seconds equal to the absolute value of *param%*.
0 < *param%* < 600	LCD is on; system sleeps for *param%* seconds.
param% ≥ 600	LCD is on; system idles (no sleep-wake cycle) for *param%* milliseconds (ms).

In all cases, the system resumes operation if a key is pressed.

• **Example 2**

```
SLEEP 0       'LCD is off; system waits for a keypress

SLEEP -10     'LCD is off; system sleeps for 10 seconds or
              'until a keypress

SLEEP 10      'LCD is on; system sleeps for 10 seconds or
              'until a keypress

SLEEP 850     'LCD is on; system idles for 850 ms or until a
              'keypress
```

SOUND Statement

• Action

Generates a sound through the speaker.

• Syntax

SOUND *frequency%, duration%*

• Parameters

Argument	Description
frequency%	The desired frequency in hertz (Hz) (cycles/second). The frequency must be a numeric expression returning an integer in the range 50–8000. See the table below for the frequencies of different musical notes.
duration%	The duration of the tone. The duration must be a numeric expression returning an integer in the range of 1–2000 (2000 = 2 seconds).

Musical Note Frequencies

G#	52	E	165	C	523	G#	1661
A	55	F	175	c#	554	A	1760
A#	58	F#	185	D	587	A#	1885
B	62	G	196	D#	622	B	1976
C	65	G#	208	E	659	C	2083
c#	69	A	220	F	698	C#	2217
D	73	A#	233	F#	740	D	2349
D#	76	B	247	G	784	D#	2489
E	82	C	262	G#	831	E	2637
F	87	c#	277	A	880	F	2794
F#	93	D	294	A#	932	F#	2880
G	98	D#	311	B	986	G	3136
G#	104	E	330	C	1047	G#	3322
A	110	F	349	c#	1109	A	3620
A#	117	F#	370	D	1175	A#	3729
B	123	G	392	D#	1245	B	3951
C	131	G#	415	E	1329	C	4186
c#	139	A	440	F	1387		
D	147	A#	466	F#	1480		
D#	156	B	494	G	1566		

• Remarks

Do not use values outside the given ranges.

See also **BEEP**.

• Example

```
REM  SOUND Statement

SOUND 1415,100
SOUND 1440,50
SOUND 1494,50
SOUND 1523,50
SOUND 1587,50
SOUND 1698,50
SOUND 1784,50
SOUND 1880,50
SOUND 2047,50
SOUND 2329,50
SOUND 2566,50
```

See also the **FOR...NEXT** example.

STR$ Function

• **Action**

Returns a string representation of a number.

• **Syntax**

strEquiv$ = STR$ (*numeric expression%* **)**

• **Parameters**

Argument	Description
numeric expression%	Any numeric expression.

• **Remarks**

The **VAL** function complements **STR$**. See also **VAL** example.

• **Returns**

A string representation of *numeric expression%.*

• **Example**

```
REM STR$() Function

LET word$ = "Videx LaserLite"
PRINT word$
lenWord% = LEN(word$)
length$ = STR$(lenWord%)
PRINT "is "; length$; " chars.";
SLEEP 0               'pause display until keypress
```

Output:

```
Videx LaserLite
is 15 chars.
```

SWAP Statement

• Action

Exchanges the value of two variables.

• Syntax

SWAP *variable1$*, *variable2$*

or

SWAP *variable1%*, *variable2%*

• Parameters

Argument	Description
variable1, *variable2*	Can be any variable, but both variables must be of the same type.

• Remarks

Either integer or string variables can be swapped. However, the two variables must be exactly the same type or an error message appears. For example, trying to swap an integer variable with a string variable will produce an error message.

• Example

```
REM SWAP Statement
sndA% = 4800
sndB% = 3300
FOR i% = 1 TO 10
    SOUND sndA%, 500
    SWAP sndA%, sndB%
NEXT i%
```

TIME$ Function

• **Action**

Returns the current time from the operating system.

• **Syntax**

nowTime$ = TIME$ [( )]

• **Parameters**

None

• **Remarks**

None

• **Returns**

The **TIME$** function returns an eight-character string in the pattern *hh:mm:ss*, where *hh* is the hour (00–23), *mm* is the minutes (00–59), and *ss* is the seconds (00–59). A 24-hour clock is used; therefore, 8:00 PM is shown as 20:00:00.

• Example 1

```
REM TIME$() Function example 1 example 1
'Returns both date and time (in two formats).

Dim nowDate$,nowTime$,convDate$,convTime$
CLS
nowDate$ = DATE$()
nowTime$ = TIME$()
PRINT "Date is :"
PRINT nowDate$;              'standard date
SLEEP 0
GOSUB make_date
CLS
PRINT "Date is :"
PRINT convDate$;            'TimeWand I style date
SLEEP 0
CLS
PRINT "Time is :"
PRINT nowTime$;             'standard time
SLEEP 0
GOSUB make_time:
CLS
PRINT "Time is :"
PRINT convTime$;            'TimeWand I style time
SLEEP 0
END

' make_date
' description: Convert a DuraTrax OS date "MM-DD-YYYY"
' into a TimeWand I style date "YYYYMMDD"

make_date:
   nowDate$ = DATE$
   convDate$ = RIGHT$(nowDate$, 4)
   convDate$ = convDate$ + LEFT$(nowDate$, 2)
   convDate$ = convDate$ + MID$(nowDate$, 4, 2)
RETURN

' make_time
'  description: Convert a DuraTrax OS time "hh:mm:ss" into
'  a TimeWand I style time "hhmmss"

make_time:
   nowTime$ = TIME$
   convTime$ = convTime$ + LEFT$(nowTime$, 2)
   convTime$ = convTime$ + MID$(nowTime$, 4, 2)
   convTime$ = convTime$ + RIGHT$(nowTime$, 2)
RETURN
```

• Example 2

```
REM TIME$() Function example 2
CONST NL$ = CHR$(13) + CHR$(10)
T$ = TIME$
Hr = VAL(T$)
IF Hr < 12 THEN
    AMPM$ = " AM"
ELSE
    AMPM$ = " PM"
ENDIF
IF Hr > 12 THEN Hr = Hr - 12
PRINT "The time is"; NL$; STR$(Hr); RIGHT$ (T$,6); AMPM$;
SLEEP 4
```

TOKEN$ Function

• **Action**

Returns tokens from the given data file. This function allows you to search and scroll through the data file.

• **Syntax**

field$ = TOKEN$ (*filenumber%***,** *stringargument$***,** *direction%* **)**

• **Parameters**

Argument	Description
filenumber%	The file number used in the **OPEN** statement.
stringargument$	Any string expression.
direction%	Direction of search. If zero, searches forward; if not zero, searches backward.

• **Remarks**

Tokens are described as any sequence of characters that do *not* include any of the characters in the *stringargument$*.

See also **INPUT$**, **OPEN**, and **SEEK**.

The **TOKEN$** function is used to work with files in RAM. See the **CARDCMD** statement to work with files on the LaserLite Mx memory card.

• **Returns**

Returns tokens from the data file, starting at the current file position. If the last argument is non-zero, it searches backward; otherwise, it searches forward.

When there are no more tokens, " " is returned. Note that the current file position is changed after the token is read, preparing to read another

token in the same direction. The file position will be at the first non-token character that is encountered (or at the end or beginning of the file). If you want to append data to the file after calling **TOKEN$**, be sure to call **SEEK** to place the file position at the end of the file.

If **TOKEN$()** is used in an expression, the token is read into a temporary variable with a maximum size of 128. (See the discussion of **INPUT$** on page 107.)

The **TOKEN$** function is useful when searching through the data file for entries. By setting *stringargument$* to the field delimiter, **TOKEN$** returns data with the delimiters stripped out.

• **Example**

```
REM TOKEN$() Function
'Creates a tab delimited file in memory and parses it out.
myFile$ = "data.txt"
theToken1$ = CHR$(9)
theToken2$ = CHR$(13)
OPEN myFile$ FOR APPEND AS #0        'this is an output file
FOR i% = 1 TO 10               'write 10 tab delimited lines to file
    PRINT #0, "LINE "; STR$(i%) , " VIDEX DURATRAX"; STR$(i%)
NEXT i%
CLOSE #0
PRINT "Saved data to:"
PRINT myFile$;
SLEEP 0
CLS
OPEN myFile$ FOR APPEND AS #0        'this is an input file
SEEK #0,0
DO
    field1$ = TOKEN$ (0,theToken1$,0)
    'first field is delimited by a tab
    field2$ = TOKEN$ (0,theToken2$,0)
    'second field is terminated by a carriage return
    field2$ = LTRIM$ (field2$)        'trim the tab out
    CLS
    PRINT field1$
    PRINT field2$;
    SLEEP 4
LOOP UNTIL field2$ = ""
```

TOUCH Statement (for DuraTrax, LaserLite Pro, and LaserLite Mx only)

• Action

Reads or writes data to or from Touch Memory buttons capable of storing data. Note: Touch Memory buttons are also known as iButtons. (Videx BASIC) (Note: This statement is not supported by the Macintosh version of the Videx BASIC compiler.)

• Syntax

TOUCH *mode%, button_id$, start_addr%, BYTES_VAR%, STATUS_VAR%, DATA$*

• Parameters

Argument	Description
mode%	Specifies the operation to be performed. See Remarks.
button_id$	String containing the ID of the button to be accessed. The format is: **ff:xxxxxxxxxxxx**; where **f** is a hexadecimal digit of the 6-byte identification number with the more significant bytes to the left.
start_addr%	Touch Memory button address to start reading or writing.
BYTES_VAR%	Variable that returns the actual number of bytes processed and, for some modes, to specify the number of bytes to process. Do not use a constant or array element.
STATUS_VAR%	Variable that returns error codes, or to set or return the continuation byte for page-oriented (DDS) operations.
DATA$	Variable that contains information to write, or that will receive information that is read. Must be a variable for read operations.

• Remarks

The **TOUCH** statement is used to access the data area of certain Touch Memory buttons. Use the **INPUTEVT** function to simply read the ID on any Touch Memory button.

The *mode%* parameter is generally bit-oriented, with each bit representing a specific option. The following table defines the normal bit associations. Compatible options can usually be combined by adding the values of the various bits that are turned on.

Bit#	Effect when = 0	Effect when = 1	Add
0	Read	Write	1
1	Direct (no Default Data Structure)	Default Data Structure	2
2	Reserved	Reserved	4
3	ASCII string	Hexadecimal string	8
4	Reserved	Reserved	16
5	Reserved	Reserved	32
6	Reserved	Reserved	64
7	Reserved (also 8–15 reserved)		128

The format for the button ID is "**ff:xxxxxxxxx**" where **f** is a hexadecimal digit of the Family Code and **x** is a hexadecimal digit of the 6-byte identification number with the most significant bytes to the left. The colon (**:**) is to be included as shown. If this parameter is an empty string, the ID of the last accessed button is used. This is usually the button that was read in response to a button event (**INPUTEVT**). It might also be a button previously accessed for data reading or writing, if no other button IDs were read in between.

The start address is the actual memory address for direct operations or operations using the Default Data Structure (DDS). The DDS is a page-oriented packet protocol for reading and writing data to Touch Memory buttons. In the case of a DDS operation, the start address points to the length byte (or bytes) and is usually the beginning of a page.

The *BYTES_VAR%* variable needs to be set before calling **TOUCH** for direct read operations. DDS reads will determine the number of bytes from the Default Data Structure and return it in *BYTES_VAR%*. The number of characters in a hexadecimal mode data string is twice the number of bytes. Write operations determine the number of bytes to write, from the length of the *DATA$* string. After the operation, *BYTES_VAR%* is set to the actual number of bytes successfully read or written. In the case of DDS write operations, the byte number includes

the 4 or 5 overhead bytes. A read operation counts only the number of data bytes; that is it does not include the overhead bytes.

When the operation is complete, *STATUS_VAR%* contains a negative number if an error occurred, otherwise it is usually set to zero. See the error code table on page 184 for a description of the error codes. A slightly different rule applies when reading or writing a Default Data Structure (DDS). When reading a DDS, the low order byte of *STATUS_VAR%* returns the (positive) value of the continuation page byte that immediately precedes the CRC16.

For writing a DDS, the continuation byte of the written page is set to the value you have stored in *STATUS_VAR%* before the **TOUCH** statement is executed. The number of bytes stored by a DDS write operation is always four bytes more than the length of the string in *DATA$* (1 length byte, 1 continuation byte, and a 2 CRC bytes; except for a DDS with over 254 bytes, the length specification uses 2 bytes.) See Touch Memory information from Dallas Semiconductor for more details.

The *DATA$* string contains information written to or read from the button. The maximum length when using DDS mode is 510 characters in ASCII mode or 1020 characters in hexadecimal mode. Direct mode is limited only by the length of the *DATA$* string and the practicalities of the one-wire transfer process with momentary touch; however, using direct mode to read information from buttons is generally discouraged, because errors are very likely. If it is necessary to use direct mode, it is recommended that you use multiple reads with the requirement that they agree.

The *DATA$* parameter must be a **variable** unless *mode%* is a **constant** and specifies a **write** operation. If the returned data is longer than the maximum size of the specified string, it is truncated to the length of the string. For page-oriented operations, overhead bytes are added where necessary when writing, or removed and handled separately when reading. In these cases, *DATA$* contains only true data bytes. The DDS operations pass the continuation byte (the byte immediately preceding the CRC bytes) separately through *STATUS_VAR%.*

One additional use of *STATUS_VAR%* is to indicate when null characters are encountered while reading ASCII data. The high byte of *STATUS_VAR%* is usually zero, but if nulls are encountered, each one increments the high byte of *STATUS_VAR%*. See the following paragraph for more information.

Since Videx BASIC strings are null-terminated, they cannot contain a character with an ASCII value of zero. In order to work with binary data in Touch Memory buttons, the mode specification includes a bit that specifies that the data strings will represent a series of bytes in hexadecimal format. In addition, if any null bytes are encountered in a read operation in ASCII mode, the null is converted to the character with ASCII value 172 (AC hex) before it is stored in *DATA$*. This character is "¼" in the standard PC character set and is "¬" in most Windows fonts, and is unlikely to occur in most text. As an indication of this substitution, the high byte of *STATUS_VAR%* is incremented by 1 for each such byte read. The total number of data bytes in the DDS is returned in *BYTES_VAR%*. Hexadecimal mode must be used to read such data exactly. *STATUS_VAR%* also contains the continuation byte, as usual, if this is a DDS read, so the two bytes must be separated. First, take the modulo of *STATUS_VAR%* divided by 256, to get the continuation byte. (Note: Modulo arithmetic provides the remainder of an integer division, rather than the quotient. See page 18 for information on modulo arithmetic.) Subtract the modulo from *STATUS_VAR%*, and divide the result by 256 to get the number of nulls.

The table on the following page lists the possible error conditions and other configurations of the return variables. **Error -10** (button not responding) can be activated by inappropriately by an ASCII mode read of a button that has only unwritten (FF hex) bytes in the range requested. This is because this valid non-error situation cannot be distinguished from the non-response of a button different from the one whose ID is being given. If this is a possibility in your system, you must either ignore this error or devise a system solution that insures that all buttons have some information stored.

See also **HEX$** and **BIN**.

Error Codes

Mode	*STATUS_VAR%*	*BYTES_VAR%*	Condition
All	0	# read or written	Success.
DDS* read	0 to 255 (con't)	0 to 511	Success.
All	-1	0	No button or button lost contact.
All	-2	0	ID contains an illegal (non-hex) character.
All	-3	0	Button family does not support operation.
All	-4	0	Address does not exist for this family code.
All	-5	0	Some bytes out of range for family.
Hex write	-6	(# written)	Illegal hex digit in data to be written.
All read	-7	(# read)	Not enough room in destination string.
DDS read	-8	0	Bad CRC16.
All write	-9	(# written)	Unsuccessful write; retries exhausted.
All	-10	0	Button not responding; possible wrong ID.
DDS write	-11	0	String too big for DDS packet.
ASCII DDS read	Nulls/continue†	(DDS data length)	Good DDS, but null byte encountered in data.
Other ASCII read	Nulls * 256†	(# read)	Good read, but null byte encountered in data.

* Default Data Structure, a page-oriented packet protocol for Touch Memory buttons.
† Count of nulls in high byte, continuation byte (if applicable) in low byte.

• Example

```
REM TOUCH Statement
'Read the data in a Dallas Semiconductor Touch Memory button.

DIM data$ * 64, button_data$ * 100
DIM running%
DIM ibytes%, status%

running% = 1
read_mode% = 2        'we will be reading ASCII strings using
                      'the Default Data Structure event loop
WHILE running%
      CLS
      PRINT "Touch a button:" ;  'display a prompt
      INPUTEVT 0, 0, 0, 0, type%, symbol%, device%, data$
      'get an event
IF (((type% = 1) and (device% = 48)) and (left$(data$, 3) = "04:"))
'check if event was read of a 1994 button
THEN
      status% = -1 'initialize error code
      ibytes% = 0   'initialize # of bytes read
                    'data$ now contains the button ID, which
                    'we will use in the TOUCH statement to
                    'read the button data.
TOUCH read_mode%, data$, 0, ibytes%, status%, button_data$
      IF status% >= 0 THEN  'successful read of button data
            OPTION(258) = 1      'turn on the LED
            SOUND 1446, 250      'good beep
            OPTION(258) = 0      'turn off the LED
            CLS
            PRINT button_data$
            PRINT str$(ibytes%); " bytes read";
            SLEEP 2
      ELSE
            CLS
            PRINT "Button read"
            PRINT "failed!";
            SOUND 698, 250       'sound a low beep
            SLEEP 2
      ENDIF
ENDIF
WEND
```

UCASE$ Function

• **Action**

Returns a string identical to the parameter string, but with all letters
converted to uppercase.

• **Syntax**

word$ = UCASE$ (*stringexpression$* **)**

• **Parameters**

Argument	Description
stringexpression$	The string to convert to uppercase. This can be a string variable, string constant, or string expression.

• **Remarks**

The **UCASE$** and **LCASE$** statements are helpful in making string
comparisons case insensitive.

See also **LCASE$**.

• **Returns**

The string in uppercase letters.

• **Example**

```
REM UCASE$() Function
word$ = "UpPercAse"
PRINT word$
word$ = UCASE$(word$)
PRINT word$;
SLEEP 0
```

Output:

```
UpPercAse
UPPERCASE
```

VAL Function

• **Action**

Returns the numeric value of a string of digits.

• **Syntax**

theNum% = VAL (*stringexpression$* **)**

• **Parameters**

Argument	Description
stringexpression$	Any sequence of characters that can be interpreted as a numeric value.

• **Remarks**

The **VAL** function searches *stringexpression$* for the first digit or minus sign, and then interprets the remaining consecutive digits as the integer. The **VAL** function stops reading the string when it encounters a character that it does not recognize as a number.

The **STR$** function complements **VAL**.

• **Returns**

The numeric value of the string.

• Example

```
REM VAL() Function
startRAM$ = ENVIRON$(1)
PRINT "Using RAM..."
PRINT "Wait!";
OPEN "data.txt" FOR APPEND as #0
FOR i% = 0 TO 2000           'fill the array and eat up RAM
    PRINT #0, "12345678901234567890"
NEXT i%
EndRAM$ = ENVIRON$(1)
usedRAM% = VAL(startRAM$) - VAL(endRAM$)
'convert strings to numbers and subtract
CLS
msg$ =  "USED " + STR$(usedRAM%) + " K"
PRINT startRAM$; "  -  "; endRAM$
PRINT msg$;
SLEEP 5                 'pause display for 5 seconds or keypress
```

WHILE…WEND Statement

• Action

Executes a series of statements in a loop, as long as a given condition is true.

• Syntax

WHILE *condition*
.
.
.
WEND

• Parameters

Argument	Description
condition	An integer expression that can be translated as true (nonzero) or false (0).

• Remarks

If the *condition* argument is true (that is, if it does not equal zero), then any intervening statements are executed until the **WEND** statement is encountered. BASIC then returns to the **WHILE** statement and checks the *condition* argument. If it is still true, the process is repeated. If it is not true (or if it equals zero), execution resumes with the statement following the **WEND** statement.

Note: The **DO…LOOP** statement provides a more powerful and flexible loop control structure.

WHILE…WEND loops may be nested to any level. Each **WEND** matches the most recent **WHILE**. An unmatched **WHILE** or **WEND** statement causes an error message.

See also **DO…LOOP** and **FOR...NEXT**.

• Example

```
REM WHILE...WEND Statement
'*
'* Monitors data collector events and exits the program if
'* the power switch is turned off.
'*
CONST false = 0, true = Not false, NL$ = CHR$(13) + CHR$(10)
DIM display$ * 34
running% = true                       'set running% to true
display$ = "Waiting on you..." + NL$ + "P switch = exit"

WHILE running%                        'event loop
    CLS
    PRINT display$;                   'display a prompt
    INPUTEVT 0, 0, 0, 0, type%, symbol%, device%, data$
    'get an event
    IF type% = 6 THEN
    'if type% event is 6 then power switch was turned off
        running% = false              'set running% to false
    ELSE
        BEEP
    ENDIF
WEND

END
```

Chapter 5

BASIC Compilers

BASIC compiler programs are available for three operating platforms:
Windows 95/98/NT, DOS, and Macintosh. Both the Windows and
Macintosh compilers provide a drag-and-drop interface. The DOS
version is a command line program with parameters.

The compiler program for Windows is **Vxbasicw.exe**.
The compiler program for DOS is **Vxbasic.exe**.
The compiler program for Macintosh is **Videx BASIC**.

Vxbasicw.exe Overview for Windows

The compiler program for Windows 95/98/NT is **Vxbasicw.exe**.

Vxbasicw.exe for Windows is an executable program. It was developed
to demonstrate the functionality of **Vxbasic.dll**. You may use the **Run**
command from the Windows **Start** menu to pass parameters to
Vxbasicw.exe. Parameters and defaults are identical to the DOS
Vxbasic.exe program. Alternatively, you may start the program by
dragging-and-dropping a source code file onto the **Vxbasicw.exe** icon.

To execute, **Vxbasicw.exe** requires the **Vxbasic.dll** file. The file may be
installed either in the directory with **Vxbasicw.exe** or in the system path
for DLL's.

Windows DLL
A 32-bit dynamically linked library (DLL) for compiling your BASIC
source code is included with the Application Builder software. The
header file, **Vxbasic.h**, documents the calling convention.

Vxbasic.exe Overview for DOS

The compiler program for DOS computers is **Vxbasic.exe**.

This is a command line program with parameters. **Vxbasic.exe** has the following syntax:

```
vxbasic <input filename.ext>
```

By convention, you should use **.B** as the input file extension and **.S** as the output file extension. If there are any errors in the code, the compiler quits at the first error encountered and displays the error and source code line on the screen.

Videx BASIC Overview for Macintosh

The compiler program for Macintosh computers is **Videx BASIC**.

Videx BASIC is an AppleScript server that can accept **Open Document** events. If there are any errors in the code, the compiler quits at the first error encountered and displays the error and source code line on the screen.

Appendix A

BASIC Reserved Words

The following is a list of BASIC reserved words:

ABS	HEX$	SEEK
AND	IF	SEEKH
APPEND	INKEY$	SLEEP
AS	INPUT$	SGN
ASC	INPUTEVT	SOUND
BEEP	INSTR	STEP
BIN	LCASE$	STR$
CARDCMD	LEFT$	SWAP
CARDSTATUS	LEN	THEN
CHR$	LET	TIME$
CLOSE	LOCATE	TOKEN$
CLS	LOF	UCASE$
COMMCLOSE	LOFH	UNTIL
COMMINPUT	LOOK$	VAL
COMMOPEN	LOOKUP	WEND
COMMPRINT	LOOP	WHILE
CONST	LTRIM$	
DATE$	MID$	
DIM	MOD	
DO	NEXT	
ELSE	NOT	
ELSEIF	ON	
END	OPEN	
ENDIF	OPTION	
ENVIRON$	OR	
EOF	PATTERN	
ERR	PRINT	
EXIT	REM	
FOR	RETURN	
GOSUB	RIGHT$	
GOTO	RTRIM$	

Notes:

Appendix B

LaserLite Mx Modulus Information

(Notes about hashed indexes on the LaserLite Mx)

Hash tables differ from other tables or arrays because they provide the
LaserLite Mx system a fast way to seek data in an ASCII data file. They
provide nonsequential access to data elements through the use of a hash
function that converts the key field into an integer and divides it by the
size (modulus) of the hash table. The remainder becomes the "key,"
"lookup value," or "bin" that indexes where to find the data element.
The LaserLite Mx system provides a built-in hash function for
distributing data elements into a hash table.

While the actual hash algorithm for the LaserLite Mx is written in 8051
assembler, it is based on the following C language code example which
is based on Allen Holub's portable adaptation of Peter Weinberger's
generic hashing algorithm.

```
/*---Hash PJW--------------------------------------
/*An adaptation of Peter Weinberger's (PJW) generic
/*hashing algorithm based on Allen Holub's version.
/*Accepts a pointer to a datum to be hashed and
/*returns an unsigned integer.
/*------------------------------------------------*/

#include <limits.h>
#define BITS_IN_int      ( sizeof(int) * CHAR_BIT )
#define THREE_QUARTERS   ((int)((BITS_IN_int * 3) / 4))
#define ONE_EIGHTH       ((int)(BITS_IN_int / 8))
#define HIGH_BITS ( ~((unsigned int)(~0) >> ONE_EIGHTH ))
unsigned int HashPJW ( const char * datum )
{
  unsigned int hash_value, i;
  for ( hash_value = 0; *datum; ++datum )
  {
    hash_value = ( hash_value << ONE_EIGHTH ) + *datum;
    if (( I = hash_value & HIGH_BITS ) ! = 0 )
      hash_value =
        ( hash_value ^ ( I >> THREE_QUARTERS 00 &
          ~HIGH_BITS;
  }
  return 9 hash_value );
}
```

What Modulus should be used when creating an Indexed File?

Use the following guidelines to select a modulus for LaserLite Mx indexed files:

- Optimal trade-off between performance and space efficiency is achieved when the anticipated number of records is less than twice the modulus.

- A prime number modulus provide the best distribution of data. The following prime numbers selections can provide optimal distribution:

  13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 21701, 23209, 44497.

- The LaserLite Mx can support a modulus up to 65535.

Appendix C

BASIC Sample Programs

This appendix contains two BASIC programs: **Default.b** and
DEMOMX.b. **Default.b** is the default application that is loaded with the
operating system if another application is not specified. **DEMOMX.b** is
a demo application for the LaserLite Mx. It contains the subroutines
process_cardcmd_error and **process_card_error** that are referred to in
the **CARDCMD** statement examples.

Default.b

```
'* This is the default application which is loaded with the
'* operating system unless another application is specified.
'
const false% = 0     'define constants
const true% = not false%
const backward% = true%
const forward% = false%

const mode_norm% = 0       'defines scrolling behavior
const mode_up% = 1
const mode_dn% = 2
const delimiter$ = chr$(13) + chr$(10)
'delimits data file lines with a carriage return/line feed
const low_voltage% = 45    'voltage which triggers warnings
const x_delay% = 150       'define delay time
dim data$ * 64             'dimension strings
dim display$ * 64
dim full_display$ * 64
dim bar$ * 64
dim bar2$ * 64
dim voltage%
dim info_index%
dim shift%
dim running%
dim mode%
dim n_times%
```

```
              sound 2793, 250          'startup beep
              sound 2637, 250
              sound 2793, 250

              open "data.txt" for append as #0  'open and name data file
              if err then gosub file_error

              data$ = ""                'initialize variables
              display$ = ""
              mode% = mode_norm%
              running% = true%
              voltage% = 60
              info_index% = 0
              shift% = 0
              while running%                      'event loop
                 gosub check_voltage       'check battery voltage
                 cls
                 print left$(display$, 16)      'display a prompt
                 print "Scan: " ;
                     inputevt 6, 1, 16, 2, type%, symbol%, device%, data$
                                       'get an event
                   on type% gosub fn_null, fn_input, fn_exit,|
                       evt_delete, fn_up, fn_down, fn_power, evt_pfail,|
                       evt_esc, fn_null, fn_null, fn_null, fn_null,|
                       fn_null, fn_null, fn_null, fn_null, fn_null,|
                       fn_left, fn_right, evt_mem, evt_bat, evt_f1,|
                       evt_f2, evt_f3, evt_f4, evt_delete
                       'process the event
              wend
              sound 1760, 250     'exit beep
              sound 1568, 250
              sound 1397, 250
              end             'return to command line


              '*
              '* fn_null
              '* description: Do nothing function.
              '*

              fn_null:
              return
```

```
'*
'* fn_input
'* description: Respond to a user input event. Beeps and
'* blinks the valid scan LED, and then appends the input to
'* the end of the data file, along with the current date
'* and time.
'*

fn_input:
    if device% = 1 then
        if len(data$) = 0 then
            'it's an ENTER key, with no data
            sound 698, 250   'sound a low beep
            return
        endif
    endif

    option(258) = 1          'turn on the LED
    sound 1446, 250          'good beep
    option(258) = 0          'turn off the LED

    seek #0, 0, e          'set file position to end of file
    print #0, date$ ; " " ; time$ ; " " ; data$
    'write input, time & date to the file
    if err then gosub file_error

    display$ = data$         'display input on screen
    mode = mode_dn%          'same as scrolling down to
                             'last line of file
    full_display$ = data$    'initialize for scroll right
                             '& left
    shift% = 0
return


'*
'* fn_exit
'* description: Respond to an exit event. Exit events are
'* generated when an unlock command is sent to the unit
'* during an INPUTEVT statement. The typical response is to
'* end the program, returning to the command line.
'*

fn_exit:
        running% = false%
return
```

```
'*
'* fn_up
'* description: Respond to a scroll-up event. These events
'* are generated when the scroll-up key is pressed.
'* Respond by getting an input token from the data file,
'* and displaying it on the screen.
'*

fn_up:
        'skip the last token displayed, if
        'we were scrolling down or scanning
if mode% = mode_dn% then
    bar$ = token$(0, delimiter$, backward%)
endif

display$ = token$(0, delimiter$, backward%)  'get next line
if display$ = "" then
    display$ = "-TOP OF DATA-"
    mode = mode_norm%
else
    display$ = mid$(display$, 21)  'remove date and time
    mode% = mode_up%
endif

sound 1397, 10   'notify user that we got the key press

full_display$ = display$   'initialize scroll right & left
    shift% = 0
return
```

```
'*
'* fn_down
'* description: Respond to a scroll-down event. These
'* events are generated when the scroll-down key is
'* pressed. Respond by getting an input token from the data
'* file, and displaying it on the screen.
'*

fn_down:
    'skip the last token displayed, if we were scrolling up
if mode% = mode_up% then
    bar$ = token$(0, delimiter$, forward%)
endif
display$ = token$(0, delimiter$, forward%)
    'get the next line if we're already at the bottom
    'of the file, display system information
if display$ = "" then
    gosub scroll_info
    mode% = mode_norm%
else
        'remove the date and time
    display$ = mid$(display$, 21)
    mode% = mode_dn%
endif
sound 1397, 10   'notify user that we got the keypress
full_display$ = display$ 'initialize for scroll right & left
shift% = 0
return


'*
'* fn_left
'* description: Scroll left key was pressed (LaserLite
'* Pro/Mx). Scroll to the left to display a long entry.
'*

fn_left:
sound 1397, 10   'click so the user knows we got the key
'check whether we have scrolled all the way
'off the display otherwise, increment position
if shift% < len (full_display$) then
    shift% = shift% + 1
endif
display$ = ""    'build display one character at a time
for foo% = 1 to 16
    if len (full_display$) >= (foo% + shift%) then
    display$ = display$ + mid$(full_display$, foo%+shift%, 1)
    endif
next foo%
return
```

```
'*
'* fn_right
'* description: Scroll right key was pressed (LaserLite
'* Pro/Mx only). Scroll right to display a long entry.
'*

fn_right:
sound 1397, 10    'click so the user knows we got the key
if shift% > 0 then      'check if we have scrolled back to
    shift% = shift% - 1    'the beginning of the display,
endif                       'otherwise decrement position

'build display one character at a time
display$ = ""
for foo% = 1 to 16
    if len (full_display$) >= (foo% + shift%) then
      display$ = display$ + mid$(full_display$, foo%+shift%, 1)
    endif
next foo%
return


'*
'* fn_power
'* description: Respond to a power event. These events are
'* generated when the power switch is moved to the off
'* position. Respond by setting the hardware into a
'* low-power state.
'*

fn_power:
option(256) = 0      'turn off the LCD
sleep 0              'sleep indefinitely
option(256) = 1      'turn the LCD back on
return


'*
'* file_error
'* description: There was an error opening or writing to a
'* file. Alert the user.
'*

file_error:
gosub ring
cls
print "Error accessing"
print "data file!";
sleep 10
return
```

```
'*
'* evt_esc
'* description: Escape key was pressed (LaserLite Pro/Mx).
'*

evt_esc:
    sound 1397, 10   'click so the user knows we got the key
    cls
    print "Escape key"
    print "was pressed.";
    n_times% = 1
    gosub kill_time
return


'*
'* evt_mem
'* description: Memory key was pressed (LaserLite Pro/Mx).
'* Display remaining available RAM.
'*

evt_mem:
    sound 1397, 10   'click so the user knows we got the key
    cls
    print environ$(1); " free";
    n_times% = 2
    gosub kill_time
return


'*
'* evt_bat
'* description: Battery key was pressed (LaserLite Pro/Mx).
'* Display battery voltage.
'*

evt_bat:
    sound 1397, 10   'click so the user knows we got the key
    cls
    print environ$(0); " volts";
    n_times% = 2
    gosub kill_time
return
```

```
'*
'* evt_f1
'* description: F1 key was pressed (LaserLite Pro/Mx).
'*

evt_f1:
    sound 1397, 10   'click, so the user knows we got the key
    cls
    print "F1 key was"
    print "pressed.";
    n_times% = 1
    gosub kill_time
return

'*
'* evt_f2
'* description: F2 key was pressed (LaserLite Pro/Mx).
'*

evt_f2:
    sound 1397, 10   'click, so the user knows we got the key
    cls
    print "F2 key was"
    print "pressed.";
    n_times% = 1
    gosub kill_time
return
```

```
'*
'* evt_f3
'* description: F3 key was pressed (LaserLite Pro/Mx).
'*

evt_f3:
    sound 1397, 10   'click, so the user knows we got the key
    cls
    print "F3 key was"
    print "pressed.";
    n_times% = 1
    gosub kill_time
return


'*
'* evt_f4
'* description: F4 key was pressed (LaserLite Pro/Mx).
'* Display operating system.
'*

evt_f4:
    sound 1397, 10   'click, so the user knows we got the key
    bar$ = environ$(2)
    cls
    print left$ (bar$, 16)
    if len(bar$) > 16 then
        print right$ (bar$, len(bar$)-16);
    endif
    n_times% = 2
    gosub kill_time
return
```

```
'*  evt_delete
'*  description: F5 key was pressed (LaserLite Pro/Mx)
'*  or a 5 space barcode was scanned.
'*  Delete most recent entry.
evt_delete:
    if lof(0) = 0 then      'see if there is any data in file
        sound 2349,250
        cls
        print "There is no data"
        print "to delete!";
        n_times% = 4
        gosub kill_time
        return
    endif
    seek #0, 0, e        'move pointer to end of data file
    bar$ = token$ (0, delimiter$, backward%)
    cls
    print "Delete data?   Y"
    bar2$ = mid$ (bar$, 21)        'remove date and time
    foo% = len (bar2$)
    if foo% <= 14 then
        print bar2$;
        locate 1,14: print " N";
    else
        print ".."; right$ (bar2$, 12); " N";
    endif
    gosub questiontone
    do
        foo% = asc(inkey$())
    loop until (foo% = 2)   |        'up arrow
        or (foo% = 4) |              'down arrow
        or (foo% = 89)  |              rem 'Y'
        or (foo% = 78)  |              rem 'N'
        or (foo% = 132) |              rem MEM (Y unshifted)
        or (foo% = 42)  |              rem '*' (N unshifted)
    if (foo%=2) or (foo%=89) or (foo%=132) then
        nbytes% = len (bar$) + 2   'delete <cr/lf> characters
        gosub truncate
        gosub deletetone
        display$ = token$(0, delimiter$, backward%)
        if display$ = "" then
            display$ = "-TOP OF DATA-"
            mode = mode_norm%
        else
            display$ = mid$(display$, 21)'remove date & time
            mode% = mode_up%
        endif
        full_display$ = display$
        shift% = 0
    else
        sound 2349, 250
    endif
return
```

```
'*
'* check_voltage
'* description: Get the current voltage. If it is below
'* low_voltage% and has changed, alert the user.
'*

check_voltage:
bar$ = left$(environ$(0), 3)
foo% = (val(left$(bar$, 1)) * 10) + val(right$(bar$, 1))

    if foo% < low_voltage% then
        if foo% < voltage% then
            cls
            print "Low batteries."
            print environ$(0) ; " volts" ;
            gosub ring
            sleep 10
            voltage% = foo%
        end if
    else
        voltage% = foo%
    end if
return


'*
'* evt_pfail
'* description: It looks like power was lost when the unit
'* was not asleep. This event is generated once if the
'* unexpected loss of power can be detected.
'*

evt_pfail:
    cls
    print "Power loss,"
    print "verify data.";
    gosub ring
    sleep 0
return


'*
'* ring
'* description: Ring to alert the user of an error.
'*

ring:
    for ring_i% = 1 to 10
        sound 2093, 50
        sound 2794, 50
    next ring_i%
return
```

```
'*
'* delete tone
'* description: Tone that indicates that a record has
'* been deleted.
'*

deletetone:
    sound 3620,150
    sound 2349,250
return


'*
'* questiontone
'* description: Tone used when asking user to confirm if
'* data is to be deleted.
'*

questiontone:
        sound 2349, 150
        sound 3620, 250
return


'*
'* scroll_info
'* description: Display various information when you have
'* scrolled down to the bottom of the data file.
'*

scroll_info:
    info_index% = info_index% + 1
    if 3 < info_index% then
        info_index% = 0
    end if

    on info_index% gosub os_info, volt_info, time_info, mem_info
    return

volt_info:        'display battery voltage
    display$ = environ$(0) + " volts"
return

os_info:      'display operating system name
    display$ = environ$(2)
return

time_info:        'display time
    display$ = time$()
return

mem_info:      'display available RAM
    display$ = environ$(1) + " free"
return
```

```
'*
'* truncate
'* description: Remove nbytes% from the data file.
'*

truncate:
    if nbytes% <= 0 then
        return              'nbytes% must be positive
    end if
    high% = lofh( 0)     'get the size of the file in
    low% = lof( 0)       'high and low words
    if nbytes% <= low% then
        low% = low% - nbytes%
            'if the low word is >= nbytes%, then
            'just subtract nbytes% from low word
    elseif 0 < high% then
            'otherwise, if the high byte isn't zero,
        high% = high% - 1
            'subtract one from the high byte and set
        low% = ((low% - nbytes%) + 32767) + 1
            'the low byte to low% + 32768 - nbytes%
    else          'otherwise, the file isn't nbytes% long
        high% = 0
        low% = 0
    endif
    lof( 0) = high%, low%         'set the new file length
return


'*
'* kill time
'* description: Delays program operation while information
'* is being displayed.
'*

kill_time:
    for foo% = 1 to (n_times% * x_delay%)
    next foo%
return
```

MXDEMO.B

The following BASIC program demonstrates the functionality of a
LaserLite Mx and accesses the LaserLite Mx memory card.

```
' MXDEMO.B
' This program demonstrates the functionality of a
' LaserLite Mx. The main application writes data to a file
' and allows scrolling and deleting. In addition, another
' routine called by the <F1> key, gets a number in a
' sequence, appends it to clock data, writes it to a file on
' the memory card, reads it back and displays it. Another
' routine, invoked by pressing <F3>, offers to search for
' one of the sequence numbers in the database.
'


'define constants
    const false% = 0
    const true% = not false%
    const backward% = true%
    const forward% = false%
    const mode_norm% = 0     'defines scrolling behavior
    const mode_up% = 1
    const mode_dn% = 2
    const mode_bottom% = 3
    const mode_top% = 4
    const low_voltage% = 45     'voltage to trigger warnings
    const x_delay% = 150        'define delay time
    dim data$ * 64              'dimension strings
    dim display$ * 64
    dim full_display$ * 64
    dim bar$ * 64
    dim bar2$ * 64
    dim buffer_var$ * 256
    dim voltage%
    dim info_index%
    dim shift%
    dim running%
    dim mode%
    dim n_times%
    dim cstatus$
    dim cardreturn$ * 256
    dim crdcmd$ * 256
```

```
beginning:
    mode% = mode_norm%
    running% = true%
    voltage% = 60
    info_index% = 0
    shift% = 0
    NL$ = chr$(13) + chr$(10) 'carriage return/line feed
    sound 2793, 250                    'startup beep
    sound 2637, 250: sound 2793, 250
    open "dataram.txt" for append as #0
'open & name data file
    cls
    print "Starting"; NL$; "application....";
    cardcmd O, 2203, I, "data.txt"
    'open a data file on the memory card
    gosub process_cardcmd_error
    'report errors from sending command to card
    print #0, "cardresult% = "; str$(cardresult%)
    IF crderr% = 0 THEN
        cardresult% = cardstatus(cardreturn$)   'get results
        gosub process_card_error           'handle any errors
    ELSE
        goto won't_work
    ENDIF
    data$ = ""
    display$ = ""
    cardcmd C, S
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
'determine file handle being used and record it in dhandle$
'so we can be sure it is the only one that allows user entry
    gosub process_card_error
    chandle$ = mid$(cardreturn$,14,2)
    dhandle$ = chandle$                'initialize variables
    while running%                     'event loop
        gosub check_voltage           'check battery voltage
        cls
        print left$(display$, 16)  'display a prompt
        print "Scan: ";
        inputevt 6, 1, 16, 2, type%, symbol%, device%, data$
            'get event
        on type% gosub fn_null, fn_input, fn_exit,|
            evt_delete, fn_up, fn_down, fn_power, evt_pfail,|
            evt_esc, fn_null, evt_nomodule, evt_nocard,|
            evt_badcard, evt_cardID, evt_card_interrupted,|
            fn_null, fn_null, fn_null, fn_left, fn_right,|
            evt_mem, evt_bat, evt_f1, evt_f2, evt_f3, evt_f4,|
            evt_delete                  'process the event
    wend
    sound 1760, 200      'exit beep
    sound 1568, 200
    sound 1397, 200
end                      'return to command line
```

```
'*
'* fn_null
'* description: Do nothing function.
'*
fn_null:
return

'*
'* fn_input
'* description: Respond to a user input event by beeping
'* and blinking the valid scan LED, and appending the input
'* to the end of the data file along with the current date
'* and time.
'*

fn_input:
    if device% = 1 then
        if len(data$) = 0 then
            'it's an ENTER key, with no data
            sound 698, 250            'sound a low beep
            return
        endif
    endif
    cardcmd F%, data$        'search for data$ in data file
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
    if (cardresult% = 35) then   'data not found in data file
        option(258) = 1       'turn on the LED
        sound 1446, 250       'good beep
        option(258) = 0       'turn off the LED
        record$ = data$ + chr$(09) + time$ + chr$(09) + date$
            'build a record with fields that are tab delimited
        IF dhandle$ <> chandle$ THEN gosub just_open
            'be sure we are going to write to correct file
        cardcmd A, record$            'append the record
        gosub process_cardcmd_error
        cardresult% = cardstatus(cardreturn$)
        gosub process_card_error
        display$ = data$     'display the input on the screen
        mode = mode_dn%       'same as if we had just scrolled
                              'down to the last line of file
        full_display$ = data$
            'initialize for scroll right & left
        shift% = 0
    elseif (cardresult% <> 0) then
        gosub process_card_error
    else                      'duplicate scan, do not accept it
        cls
        locate 0,0
        print data$
        print "already scanned";
        gosub card_error_tone
        'n_times% = 1
        'gosub kill_time
```

```
        sleep 1000
    endif
return


'*
'* fn_exit
'* description: Respond to an exit event. Exit events are
'* generated when an unlock command is sent during an
'* INPUTEVT statement. The typical response is to end the
'* BASIC program, returning to the command line.
'*

fn_exit:
        running% = false%
return
```

```
'*
'* fn_up
'* description: Respond to a scroll-up event. These events
'* are generated when the scroll-up key is pressed.
'*

fn_up:
    sound 1397, 10    'click so user knows we got the key

    if (mode% = mode_bottom%) OR (mode% = mode_down%) then
        'if we have been scrolling down or we're at the
        'bottom already, then fetch the last record.

        cardcmd M
        gosub process_cardcmd_error
        cardresult% = cardstatus(cardreturn$)
        gosub process_card_error
        mode% = mode_norm%
    else

        cardcmd M, 1, R      'otherwise move back one record
        gosub process_cardcmd_error
        cardresult% = cardstatus(cardreturn$)
        gosub process_card_error
    endif

    if cardresult% = 37 then            'top of file
        display$ = " -TOP OF DATA-"
        mode% = mode_top%

    elseif cardresult% = 23 then        'no data
        display$ = "-DATA FILE EMPTY"
        mode% = mode_top%

    else                                'otherwise get the record
        display$ = cardreturn$
        mode% = mode_up%
    endif

                'prepare for shift left and right
    full_display$ = display$
    shift% = 0
return
```

```
'*
'* fn_down
'* description: Respond to a scroll-down event. These
'* events are generated when the scroll-down key is
'* pressed.
'*

fn_down:
    sound 1397, 10   'click so the user knows we got the key

    if (mode% = mode_top%) then    'we are at top of file
                                   'so don't move pointer
        cardcmd M
        gosub process_cardcmd_error
        cardresult% = cardstatus(cardreturn$)
        gosub process_card_error
            'get the next line (if not at the top of the file)
    else
        cardcmd M, 1, F
        gosub process_cardcmd_error
        cardresult% = cardstatus(cardreturn$)
        gosub process_card_error
    endif

    if cardresult% = 36 then
        gosub scroll_info
            'if at end of file, display system info
        mode = mode_bottom%

    elseif cardresult% = 23 then
        gosub scroll_info
        mode = mode_bottom%

    else
        display$ = cardreturn$
        mode% = mode_dn%
    endif

    full_display$ = display$
            'initialize for scroll right & left
    shift% = 0
return
```

```
'*
'* fn_left
'* description: Scroll left key was pressed (LaserLite
'* Pro/MX). Scroll to the left to display a long entry.
'*

fn_left:
    sound 1397, 10    'click so the user knows we got the key
                'check if we have scrolled all the way off of
                'the display otherwise, increment position

    if shift% < len (full_display$) then
        shift% = shift% + 1
    endif

                'build display one character at a time
    display$ = ""
    for foo% = 1 to 16
        if len (full_display$) >= (foo% + shift%) then
        display$ = display$ + mid$(full_display$, foo% + shift%, 1)
        endif
    next foo%
return


'*
'* fn_right
'* description: Scroll right key was pressed (LaserLite
'* Pro/MX). Scroll to the right to display a long entry.
'*

fn_right:
    sound 1397, 10    'click so the user knows we got the key
                'check if we have scrolled all the way back
                'to the beginning of the display otherwise
                'decrement position
    if shift% > 0 then
        shift% = shift% - 1
    endif

    display$ = ""    'build display one character at a time
    for foo% = 1 to 16
        if len (full_display$) >= (foo% + shift%) then
         display$ = display$ + mid$(full_display$, foo%+shift%, 1)
        endif
    next foo%
return
```

```
'*
'* fn_power
'* description: Respond to a power event. These events are
'* generated when the power switch is moved to the off
'* position. Respond by setting the hardware to a low-power
'* state.
'*

fn_power:
'option(256) = 0     'turn off the LCD
sleep 0                  'sleep indefinitely
'option(256) = 1     'turn the LCD back on
return



'*
'* file_error
'* description: There was an error opening or writing to a
'* file. Alert the user.
'*

file_error:
    gosub ring
    cls
    print "Error accessing "
    print "data file!"
    sleep 10
return



'*
'* evt_esc
'* description: Escape key was pressed (LaserLite Pro/Mx).
'*

evt_esc:
    sound 1397, 10   'click so user knows we got the key
    cls
    print "Escape key"
    print "was pressed.";
    'n_times% = 1
    'gosub kill_time
    sleep 1000
return
```

```
'*
'* evt_mem
'* description: Memory key was pressed (LaserLite Pro/Mx).
'* Display remaining available RAM.
'*


evt_mem:

    sound 1397, 10      'click so user knows we got the key

retry_mem:
    cardcmd C, F
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
    gosub process_card_error

    card_space$ = mid$ (cardreturn$, 10, 4) 'avail space (hex)
    total_files$ = mid$ (cardreturn$, 14, 2) '# of files (hex)

    foo% = bin (card_space$, 4)    'convert hex to integer
    card_space$ = str$ (foo%)      'convert integer to string

    foo% = bin (total_files$, 2)
    total_files$ = str$ (foo%)

    cls
    print total_files$; " files on card" 'display available
    print card_space$; "K available";    'space on card
    'n_times% = 3
    'gosub kill_time
    sleep 3000
return


'*
'* evt_bat
'* description: Battery key was pressed (LaserLite Pro/Mx).
'* Display battery voltage.
'*

evt_bat:
    sound 1397, 10      'click so user knows we got the key
    cls
    print environ$(0); " volts";
    sleep 3000
return
```

```
'*
'* evt_f1
'* description: F1 key was pressed (LaserLite Pro/Mx).
'*

evt_f1:
    'sound 1397, 10  'click so user knows we got the key
    cls
    gosub rwloop
return


'*
'* evt_f2
'* description: Gets the system restarted.
'*

evt_f2:
    sound 1397, 10    'click
    cls
    print "Updating card..."

just_open:            'This routine opens the data file for
                      'appending user input
    cardcmd O, 2203, I, "data.txt"
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
    gosub process_card_error
    cardcmd C, S          'makes sure that we know this is the
                          'file to use for user input
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
    gosub process_card_error
    chandle$ = mid$(cardreturn$,14,2)
    dhandle$ = chandle$
return
```

```
'*
'* evt_f3
'* description: This is a routine for searching records
'* written to the numbers file. It presents an input field
'* into which the user may key in up to a five character
'* number. Then the program searches for that number.
'* It is a demonstration of lookup for records written on
'* the fly.
'*

evt_f3:
    sound 1397, 10   'click so user knows we got the key
    cls
    locate 0,0
    print "Opening file on"
    print "card.";
    gosub questiontone
    cardcmd O, 2203, I, "rwloop2.txt"
                        'This is numbers database
    gosub process_cardcmd_error
    IF crderr% THEN return
    cardresult% = cardstatus(cardreturn$)
    gosub process_card_error
    cardcmd C, S
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
    gosub process_card_error
    chandle$ = mid$(cardreturn$,14,2)
          'Record current file handle
    seekprompt$ = "Number to seek"
                            'Set up prompt for the user
    looping% = true%                'Set up an inputevt loop
    cls
    while looping%
       gosub check_voltage        'check battery voltage
       cls
       print seekprompt$; NL$; "-> ";    'display a prompt
       inputevt 4, 1, 10, 2, type%, symbol%, device%, data$
                        'get an event
       on type% gosub fn_null, fn_seek, fn_quitseek,|
          fn_quitseek, fn_quitseek, fn_quitseek, fn_power,|
          evt_pfail, fn_quitseek, fn_null, evt_nomodule,|
          evt_nocard, evt_badcard, evt_cardID,|
          evt_card_interrupted, fn_null, fn_null, fn_null,|
          fn_left, fn_right, evt_mem, evt_bat, fn_quitseek,|
          fn_quitseek, fn_quitseek, fn_quitseek, fn_quitseek
                  'process the event
                  'Some keys will quit the function. Only the
                  'Enter key with data will perform the seek.
    wend
    gosub deletetone              'exit tone
return
```

```
'*
'* fn_seek
'* description: Demo for lookup in the numbers file.
'*

fn_seek:
    cardcmd F, data$
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
    IF cardresult% = 0 THEN
        sound 1397, 10
        rtime$ = mid$(cardreturn$,instr(cardreturn$,|
chr$(09))+1,8)
        rdate$ = right$(cardreturn$,5)
        num$ = left$(cardreturn$,instr(cardreturn$,chr$(09))-1)
        cls
        locate 0,0
        print "Found! "; num$
        print rdate$; " @ "; rtime$;
        device% = 0
        while device% <> 1
            inputevt 0,0,0,0, type%, symbology%, device%, data$
        wend
        sound 1397, 10
    ELSEIF cardresult% = 35 THEN
        sound 698, 700
        cls
        locate 0,0
        print data$; " not in file..."
        device% = 0
        while device% <> 1
            inputevt 0,0,0,0, type%, symbology%, device%, data$
        wend
        sound 1397, 10
    ELSE
        gosub process_card_error
        looping% = false%
    ENDIF
return
```

```
'*
'* fn_quitseek
'* description: Get out of the seek loop.
'*

fn_quitseek:
    looping = false%
return


'*
'* evt_f4
'* description: F4 key was pressed (LaserLite Pro/Mx).
'* Display operating system.
'*

evt_f4:
'   optnum = 255
'   optval = 0
'   do
'       sound 1397, 10   'click so user knows we got the key
'       cls
'       print "option("; optnum ;")=";optval
'       print "new #: ";
'       inputevt 8,1,11,2,opt_type,opt_sym,opt_dev,opt_data$
'       if opt_type = 1 then
'           optnum = val(opt_data$)
'           optval = option(optnum)
'       endif
'       if optnum = 0 then
'           print opt_type, opt_data$
'           gosub kill_time
'       endif
'   loop while opt_type = 1
    'bar$ = environ$(2)+ " " + left$(environ$(4),5) + " " +|
right$(environ$(4),5)
    if vertoggle = 0 then
        bar$ = environ$(2)
        vertoggle = 1
    else
        bar$ = environ$(4)
        vertoggle = 0
    endif
    cls
    print left$ (bar$, 16)
    if len(bar$) > 16 then
        print right$ (bar$, len(bar$)-16);
        'print right$ (bar$,16);
    endif
    'n_times% = 2
    'gosub kill_time
    sleep 2000
return
```

```
'*
'* evt_delete
'* description: F5 key was pressed or a barcode consisting
'* of 5 spaces was scanned. Delete the current record.
'* The current record may be the last record when appending
'* or the record being viewed when scrolling.
'*

evt_delete:

    cardcmd M                   'get the current record
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
    gosub process_card_error

    IF cardresult% = 23 THEN            '23 means no data
        sound 2349,250
        cls
        print "There is no data"
        print "to delete!";
        n_times% = 3
        gosub kill_time
        display$ = ""
        return
    ENDIF

    bar2$ = cardreturn$             'make the user confirm
    cls
    print "Delete data?   Y"
    foo% = len (bar2$)
    if foo% <= 14 then
        print bar2$;
        locate 1,14
        print " N";
    else
        print ".."; left$ (bar2$, 12); " N";
    endif
    gosub questiontone
```

```
    do
        foo% = asc(inkey$())
    loop until (foo% = 2)   |       rem up arrow
        or (foo% = 4)|              rem down arrow
        or (foo% = 89)   |         rem 'Y'
        or (foo% = 78)   |         rem 'N'
        or (foo% = 132)  |         rem MEM (Y unshifted)
        or (foo% = 42)   |         rem '*' (N unshifted)

    if (foo%=2) or (foo%=89) or (foo%=132) then
        'go ahead and delete (hide) the record at pointer
        cardcmd M, H
        gosub process_cardcmd_error
        csvalid% = 0
        cardresult% = cardstatus(cardreturn$)
        gosub process_card_error

        gosub deletetone
        cls
        locate 0,0
        print "..record deleted"

        cardcmd M            'bring up the next record
        gosub process_cardcmd_error
        cardresult% = cardstatus(cardreturn$)
        gosub process_card_error

        cls                  'prepare the display
        display$ = cardreturn$

        IF cardresult% = 23 THEN
            'if no more records then its empty
            display$ = "-DATA FILE EMPTY-"
            mode = mode_norm%
        ENDIF

        full_display$ = display$
            'initialize for scroll left & right
        shift% = 0

    else
        sound 2349, 250
    endif
return
```

```
'*
'* check_voltage
'* description: Get the current voltage. If it is below
'* low_voltage% and has changed, alert the user.
'*

check_voltage:
    bar$ = left$(environ$(0), 3)
    foo% = (val(left$(bar$, 1)) * 10) + val(right$( bar$, 1))
    if foo% < low_voltage% then
        if foo% < voltage% then
            cls
            print "Low batteries."
            print environ$(0) ; " volts" ;
            gosub ring
            sleep 10000
            voltage% = foo%
        end if
    else
        voltage% = foo%
    end if
return


'*
'* evt_pfail
'* description: It looks like power was lost when the unit
'* was not asleep. This event is generated once if the
'* unexpected loss of power can be detected.
'*

evt_pfail:
    cls
    print "Power loss,"
    print "verify data.";
    gosub ring
    sleep 10000
return
```

```
'*
'* ring
'* description: Ring to alert the user of an error.
'*

ring:
    for ring_i% = 1 to 10
        sound 2093, 50
        sound 2794, 50
    next ring_i%
return



'*
'* delete tone
'* description: Tone that indicates that a record has been
'* deleted.
'*

deletetone:
        sound 3620,150
        sound 2349,250
return


'*
'* card_error_tone
'* description: Tone indicating a card command error.
'*

card_error_tone:
        sound 2349, 300
        sound 1885, 600
return



'*
'* questiontone
'* description: Tone used when asking user to confirm if
'* data is to be deleted.
'*

questiontone:
        sound 2349, 150
        sound 3620, 250
return
```

```
'*
'* scroll_info
'* description: Display various information when you have
'* scrolled down to the bottom of the data file.
'*

scroll_info:
    info_index% = info_index% + 1
    if 6 < info_index% then
        info_index% = 0
    end if

    on info_index% gosub os_info, volt_info, time_info,|
date_info, mem_info, cardfile_info, cardmem_info
    return

volt_info:       'display battery voltage
        display$ = environ$(0) + " volts"
return

os_info:         'display operating system name
        display$ = environ$(2)
return

time_info:       'display time
        display$ = time$()
return

date_info:       'display current date
        display$ = date$()
return

mem_info:      'display available RAM
        display$ = environ$(1) + " RAM free"
return

cardfile_info:        'call card memory routine
    cardcmd C, F
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
    gosub process_card_error
    total_files$ = mid$ (cardreturn$, 14, 2)
        '# of files (hex)
    foo% = bin (total_files$, 2)  'convert hex to integer
    total_files$ = str$ (foo%)    'convert integer to string
    cls
    display$ = total_files$ + " files on card"
                            'display number of files on card

return
```

```
cardmem_info:     'call card memory routine
    cardcmd C, F
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
    gosub process_card_error
    card_space$ = mid$ (cardreturn$, 10, 4)
                                  'available space (hex)
    foo% = bin (card_space$, 4)   'convert hex to integer
    card_space$ = str$ (foo%)     'convert integer to string
    cls
    display$ = card_space$ + "K available"
                              'display available space on card
return


'*
'* truncate
'* description: Remove nbytes% from the data file (not
'* used with Mx data files).
'*

truncate:
    if nbytes% <= 0 then     'nbytes% must be positive
        return
    end if
    high% = lofh(0)          'get the size of the file in
    low% = lof(0)            'high and low words
    if nbytes% <= low% then
        low% = low% - nbytes%
            'if the low word is >= nbytes%, then
            'just subtract nbytes% from low word
    elseif 0 < high% then
        high% = high% - 1
        low% = ((low% - nbytes%) + 32767) + 1
            'otherwise, if the high byte isn't zero,
            'subtract one from the high byte and set
            'the low byte to low% + 32768 - nbytes%
            'otherwise, the file isn't nbytes% long
    else
        high% = 0
        low% = 0
    endif
    lof(0) = high%, low%           'set the new file length

return
```

```
'*
'* kill_time
'* description: Delays program operation while information
'* is being displayed.
'*

kill_time:
    for foo% = 1 to (n_times% * x_delay%)
    next foo%
return


'*
'* rwloop
'* description: Keeps an increasing number count (starting
'* at 1).Reads system date and time to a memory variable.
'* Writes a record to the memory card. Reads the record
'* out. Displays it on the screen. Escape key exits the
'* loop.
'*

rwloop:
    sound 1397, 10   'click so user knows we got the key
    cls
    locate 0,0
    print "Opening file on"
    print "SSFDC.";
    gosub questiontone
    gosub init_card
    cardcmd O, 2203, I, "rwloop2.txt"
                                'open the numbers data file
    gosub process_cardcmd_error
    IF crderr% THEN return
    cardresult% = cardstatus(cardreturn$)
    gosub process_card_error

    cardcmd C, S              'record the current file handle
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
    gosub process_card_error
    chandle$ = mid$(cardreturn$,14,2)

    cardcmd M                 'Get the current record
    gosub process_cardcmd_error
    cardresult% = cardstatus(cardreturn$)
    gosub process_card_error
    IF cardresult% = 23 THEN      'If file is empty, then
        loopcount% = 1            'we'll start at 1.
                                  'Otherwise we'll increment
    ELSE                          'the last number written
        loopcount% = |
val(left$(cardreturn$,instr(cardreturn$,chr$(09))-1)) + 1
    ENDIF
```

```
   WHILE running% and (27 <> ASC(INKEY$())) AND (crderr% = 0)
            'set up a loop where escape key exits
      nowdate$ = left$(DATE$(),5)
      'fetch date and time from system and build a record
      nowtime$ = TIME$()
      cardrecord$ = str$(loopcount%) + chr$(09) + nowtime$|
+ chr$(09) + nowdate$
      cardcmd A, cardrecord$                  'add it
      IF crderr% THEN return
      gosub process_cardcmd_error
      cardresult% = cardstatus(cardreturn$)
      gosub process_card_error
      cardcmd M, -1, F           'move to bottom of file
      gosub process_cardcmd_error
      cardresult% = cardstatus(cardreturn$)  'get record
      gosub process_card_error
      timewritten$ = cardreturn$
      locate 0,0
      print "Record Read....."
      print timewritten$;                'then display it
      loopcount% = loopcount% + 1     'increment number
   WEND
   gosub deletetone     'click so user knows we got the key
return


'*
'* send_card_cmd
'* description: Go to the routine that initializes the card
'* and updates its status then send the card command string
'* to the card. Clear the card status valid flag before
'* returning.
'*

send_card_cmd:
   cardcmd crdcmd$
   gosub process_cardcmd_error
   csvalid% = 0
return
```

```
'*
'* init_card
'* description: Figures out what state the card module is
'* in and sets it in the last known state. If the Mx has
'* gone to sleep since the last command issued, then the
'* memory card software must be rebooted. In future
'* releases, this will be handled automatically by the
'* operating system.If the application restarted since the
'* last card command, then it is not necessary to update
'* card status.
'*

init_card:
    cardcmd V      'The version number tells us what program
                   'is running (whether the card is booted)
    gosub process_cardcmd_error
    cardresult% = cardstatus(buffer_var$)
    IF "MCF" = mid$(buffer_var$,3,3) THEN
    'Firmware is running
       gosub boot_card
    ELSEIF "DMS" = mid$(buffer_var$,3,3) THEN
       'Card is already booted. No action needed.
    ELSE                    'No response. Reset it. Somebody
                            'probably pulled the card
       cardcmd -1           'reset card processor.
       gosub boot_card
    ENDIF
return


'*
'* boot_card
'* description: Boots the memory card XRAM code.
'*

boot_card:
    cardcmd ASC("B")         'Send boot command to start DMS
    cardresult% = cardstatus(buffer_var$)
                             'software in the card.
return
```

```basic
'*
'* process_card_error
'* description: Since the cardstatus() function returns an
'* integer error code, it may be quickly analyzed and
'* processed with each call.
'*

process_card_error:
    IF crderr% THEN
                'already handled by cardcmd error processing

    ELSEIF (cardresult% = 0)THEN

    ELSEIF (cardresult% = 5)THEN
        error5% = true%
        goto evt_badcard

    ELSEIF (cardresult% = 32) THEN
                        'This error means 'no file open'
    gosub card_error_tone
                'This routine assumes it was open until
    cls         'the card was removed. The system restarted
    locate 0,0    'but didn't have any information to re-open
    print "  Card Process  "  'the file and set the pointer.
    print "   interrupted  ";
    n_times% = 1
    gosub kill_time
    cls
    locate 0,0
    print "  Restarting    "
    print "  application    ";
    sleep 32767
    cls
    goto beginning

    ELSEIF (cardresult% = 37)  |          'top of file
          OR (cardresult% = 36)  |      'bottom of file
          OR (cardresult% = 23) THEN     'no data

    ELSE

error_ring:
gosub card_error_tone   'otherwise just report error number
    cls
    locate 0,0
    print "Crd module error"
    print "# "; str$(cardresult%);
    n_times% = 3
    gosub kill_time
ENDIF

return
```

```
'*
'* process_cardcmd_error
'* description: Look for an error with CARDCMD and report
'* it.
'*

process_cardcmd_error:
    crderr% = ERR
    IF crderr%  = 0 THEN                    'no error
    ELSEIF crderr% = -1 THEN
'Time-out error, perhaps someone pulled out the card or the
'memory card was busy when CARDCMD tried to issue a command.
'Remember, pulling out the card freezes the memory card
'processor. It must be reset for it to restart. That's why
'it's best to go to sleep and let the system bring it back
'up when it wakes up.
        cls
        locate 0,0
        print "No response from"
        print "  memory card!  ";
        gosub card_error_tone
        sleep 1
    ELSEIF crderr% = -2 THEN
        goto evt_nomodule
    ELSEIF crderr% = -3 THEN
        goto evt_nocard
    ELSE
        cls
        locate 0,0
        print " Cardcmd error "
        print " # ";str$(crderr%) ;
        gosub card_error_tone
        gosub kill_time
        running = false%
    ENDIF
return

'*
'* won't_work
'* description: With some errors it is best to put the unit
'* to sleep, encourage the user to fix the problem, then
'* restart the application.
'*

won't_work:
    cls
    print " Press a key to"
    print "    restart   ";
    'gosub kill_time
    sleep 10000
    end
    goto beginning
return
```

```
'*
'* evt_nomodule
'* description: This event is set when the system starts
'* and the monitor determines that there is no response
'* from a memory card module. Since this application is
'* designed to run with a memory card module, it should
'* not be allowed to run when this event is encountered.
'*

evt_nomodule:
    cls
    locate 0,0
    print " No card module"
    print "   detected    ";
    gosub card_error_tone
'A -2 error is set by the system at start up
'if it determines there is no memory card module present.
    'n_times% = 3
    'gosub kill_time
    sleep 5000
    end
return


'*
'* evt_nocard
'* description: This event is set when the system starts
'* and the monitor determines that there is no memory card
'* inserted. Since this application is designed to run with
'* a memory card, it should not be allowed to run when this
'* event is encountered. It calls a sleep routine to allow
'* the user to insert a card and restart the application.
'*

evt_nocard:
    cls
        locate 0,0
        print "    No card     "
        print "   inserted! ";
        gosub card_error_tone
        'gosub kill_time
        sleep 2000
'After waking up from sleep, the system determined that
'there was no card inserted and set a global flag to
'indicate no card. The system must wake up from sleep with a
'memory card inserted to clear this flag.
        cls
        locate 0,0
        print "Insert card then"
        print "  press a key";
        sleep 2000
        sleep 10
return
```

```
'*
'* evt_badcard
'* description: This event occurs when the system cannot
'* recognize the card or the card won't boot. If the card
'* isn't formatted, then the system sets a global flag to
'* indicate card not recognized. A partially formatted card
'* may not give the system any problem until the
'* application attempts to send it a command. Then this
'* routine may be called because of an error 05 indicating
'* that the card software system did not boot.
'*

evt_badcard:
    gosub card_error_tone
    IF error5% THEN
        n_times% = 1
        cls
        locate 0,0
        print "  Card system   "
        print "     error!     ";
        'gosub kill_time
        sleep 2000
        cls
        locate 0,0
        print "  Exiting to OS  "
        print "                ";
        'gosub kill_time
        sleep 2000
        running% = false%
    ELSE
        cls
        locate 0,0
        print "    Card not     "
        print "   recognized   ";
        'sleep 1
        'gosub kill_time
        sleep 2000
        cls
        locate 0,0
        print "  Reformat card  "
        print "   then retry    ";
        'sleep 1
        'gosub kill_time
        sleep 2000
        cls
        locate 0,0
        print "    running      "
        print "  application    ";
        'gosub kill_time
        sleep 3000
        'sleep 1
        running% = false%
    ENDIF
return
```

```
'*
'* evt_card_interrupted
'* description: Since the sleep routine saves the latest
'* status from the memory card software system, it can then
'* update the card on the next cardcmd issued after waking
'* from sleep. If there is no valid status available,
'* it indicates that an event (like a power failure)
'* interrupted the process. In either case, the application
'* is automatically restarted. This routine notifies user.
'*

evt_card_interrupted:
    cls
    locate 0,0
    print "   Card update  "
    print "   interrupted  ";
    n_times% = 1
    gosub kill_time
    cls
    locate 0,0
    print "  Application   "
    print "  restarted     ";
    n_times% = 2
    goto beginning
return


'*
'* evt_cardID
'* description: Since the last time the OS ran, the card ID
'* has changed. The OS automatically restarts the
'* application and sets this event. This routine notifies
'* user.
'*

evt_cardID:

    cls
    locate 0,0
    print "Card ID changed "
    print "                ";
    n_times% = 1
    gosub kill_time
    cls
    locate 0,0
    print "  Application    "
    print "  restarted     ";
    n_times% = 1
    gosub kill_time
    cls
return
```

Index