MS-Test Statements and Intrinsic Functions

ALLOCATE statement	Allocates a memory buffer and assigns its beginning address to a pointer variable
ASC function	Converts the numeric ASCII code for the first character in a specified string
CHDIR statement	Changes the current directory (on the current drive, or the specified drive) to a specified directory
CHDRIVE statement	Changes the default drive to the drive indicated by the first character in the specified string
CHR$ function	Converts an ASCII code into a one-character string
CLEARLIST statement	Clears the specified file list
CLIPBOARD statement	Puts a text string on the clipboard
CLIPBOARD$ function	Returns the text from the clipboard
CLOSE statement	Closes one or more files
CONST statement	Assigns a value to a constant identifier
CURDIR$ function	Returns the current directory name and the specified drive for the task
DATETIME$ function	Returns the system date and time in the format MM/DD/YY HH:MM:SS AM
DEALLOCATE statement	Frees the memory pointed to by a pointer variable and assigns NULL to the pointer variable
DECLARE statement	Declares a user-defined subroutine or function, or a subroutine or function residing in a dynamic-link library (DLL)
'$DEFINE metacommand	Defines a symbol
DIM statement	Declares variables or arrays of variables, and associates them with data types
ECHO statement	Echoes printed text to a debugging terminal (auxiliary port)
END statement	The END statement stops execution of the Test Driver script and starts "ON END" processing (if previously defined)
ENVIRON$ function	Returns the contents of the PATH environment variable
EOF function	Gives the end-of-file status of a file
ERF variable	A global string containing the filename of the file in which the last trappable run-time error occurred
ERL variable	A global integer containing the line number of the line of code that caused the last trappable run-time error
ERR variable	A global integer containing the error code of the last trappable run-time error that occurred
ERROR$ function	Returns the error message for an error code or the last trappable run-time error that occured
EXISTS function	Checks to see if a file exists
EXIT statement	Terminates execution of the current code block identified by the given keyword
FOR... NEXT statement	Repeats a block of code a specified number of times, or once for each file that exists in the file list
GLOBAL statement	Declares global variables or arrays
GOSUB statement	Jumps to the section of code specified by label
GOTO statement	Unconditionally jump to the first statement following the specified label

IF...THEN statement	Conditionally executes a block of code if the given condition is true
'$INCLUDE metacommand	Inserts the contents of a specified file in a script
INSTR function	Looks for a specified string within another string
HEX$ function	Returns a string representing the hexadecimal value of a decimal integer
KILL statement	Deletes the file(s) matching the specified file specification
LCASE$ function	Returns the contents of a specified string with all uppercase letters converted to lowercase
LEN function	Returns the length of a string
LTRIM$ function	Returns a copy of the specified string with the leading blanks removed
MID$ function	Returns a substring of specified length from a given string
MKDIR statement	Creates a new directory
NAME statement	Renames files or directories
NULL function	Generates a null pointer or value
ON END statement	Adds one or more subroutines to a list of subroutines to be called automatically as the script ends
ON ERROR statement	Lets a script trap and recover from run-time errors
OPEN statement	Opens a file
PAUSE statement	Displays a string and waits for the user to acknowledge
PRINT statement	Displays information in the viewport, or sends information to the specified file
RANDOMIZE statement	Seeds the random number generator
REALLOCATE statement	Resizes the memory buffer associated with a pointer variable
REM statement	Lets you include a comment in a script
RESUME statement	Resumes program execution when an error-trap routine is finished handling the error
RETURN statement	Returns to the statement following the most recent GOSUB statement
RMDIR statement	Removes an existing directory
RND function	Generates a pseudo-random number between 0 and 32,767
RTRIM$ function	Returns a copy of the specified string with the trailing blanks removed
RUN function	Runs a program asynchronously
RUN statement	Runs a program asynchronously or synchronously
SELECT CASE statement	Executes one of several statement blocks depending on the value of the given expression
SETFILE statement	Adds or deletes filenames from the file list
SHELL statement	Passes the specified string to the MS-DOS command processor for execution
SLEEP statement	Suspends execution of the script for a specified number of seconds or indefinitely
SPLITPATH statement	Splits the specified path name into its respective parts, copying each part to the variables provided
STATIC FUNCTION statement	Begins the definition block of a user-defined function
STATIC SUB statement	Begins the definition of a user-defined suboutine
STOP statement	Terminates execution of a script
STRING$ function	Returns a string of given size whose characters all have the given

	ASCII character code
STR$ function	Returns the ASCII string representation of an integer expression
TIMER function	Gives the number of seconds (in hundredths) since midnight
TRAP statement	Defines a block of code to be executed when the specified event (which appears in a DLL) occurs
UCASE$ function	Returns the contents of the specified string with all lowercase letters converted to uppercase
'$UNDEF metacommand	Removes a symbol from the symbol definition table that has been previously defined with the
VARPTR function	Generates a far pointer to a variable
VAL function	Returns the integer value of a specified string
VIEWPORT statement	Displays, hides, or clears the viewport window
WHILE...WEND statement	Executes a series of statements in a loop, as long as a given condition is TRUE

ALLOCATE

Description The **ALLOCATE** statement allocates a memory buffer and assigns its beginning address to *pointer-var*. **ALLOCATE** attempts to allocate a memory buffer of *num-items* * *size* bytes, where *size* is the size in bytes of the data type *pointer-var* points to. For example, if pointer-var is a **POINTER TO INTEGER**, (2 * *num-items*) bytes of memory will be allocated.

Syntax **ALLOCATE** *pointer-var, num-items*

See Also CONST, DEALLOCATE, DIM, GLOBAL, REALLOCATE

ASC

Description The **ASC** function generates the numeric ASCII code for the first character in a string.

Syntax **Ret% = ASC** (*strexp$*)

Returns A numeric value that is the ASCII code for the first character in the argument.

See Also CHR$

CHDIR

Description The **CHDIR** statement changes the current directory (on the current drive, or the drive specified) to that specified in *strexp$*.

Syntax **CHDIR** *strexp$*

Comments

The **CHDIR** statement affects only the current task. Other applications, scripts, and tasks are not affected; each maintains its own current directory information.

You can change the current directory on the current drive or another drive. For example,

```
CHDIR "\MP"
```

changes the current directory on the current drive to \MP, but

```
CHDIR "D:\MYDIR"
```

changes the current directory on the drive D: to \MYDIR, regardless of your current drive.

See Also CHDRIVE, CURDIR$, MKDIR, RMDIR

CHDRIVE

Description The **CHDRIVE** statement changes the current drive to that specified by the first character in *strexp$*.

Syntax **CHDRIVE** *strexp$*

See Also CHDIR, CURDIR$, MKDIR, RMDIR

CHR$

Description The **CHR$** function converts an ASCII code into a one-character string.

Syntax **A$ = CHR$** (*asciicode%*)

Returns A one-character string whose ASCII value is equal to *asciicode%*.

See Also ASC

CLEARLIST

Description The **CLEARLIST** statement clears the file list created by the SETFILE statement.

Syntax **CLEARLIST**

See Also SETFILE

CLIPBOARD

Description The **CLIPBOARD** statement transfers text information to the clipboard.

Syntax **CLIPBOARD [***StringExpression$* **|** CLEAR **]**

Returns If you pass a *StringExpression$*, the text is copied to the clipboard. The **CLEAR** option clears all information from the clipboard text, bitmaps, and so on.

Comments

The **CLIPBOARD** statement only supports the transfer of text to the clipboard.

See Also CLIPBOARD$

CLIPBOARD$

Description The **CLIPBOARD$** function transfers text from the clipboard.

Syntax **X$ = CLIPBOARD$**

Returns Returns the text contents of the clipboard.

Comments

The **CLIPBOARD$** function only supports the transfer of text from the clipboard. You cannot use this function to retrieve bitmaps or other types of information supported by the clipboard. If the clipboard contains both text and non-text data (such as a description and a bitmap), the **CLIPBOARD$** function only returns the text. If the clipboard contains non-text data only, the **CLIPBOARD$** function returns an empty string ("").

See Also CLIPBOARD

CLOSE

Description The **CLOSE** statement closes one or more files opened with the OPEN statement.

Syntax **CLOSE [[#]***filenumber%***][,[#]***filenumber%***]...]**

Comments

A **CLOSE** statement with no arguments closes all open files. A CLOSE statement with *filenumber%* between 1 and 5 closes the file associated with *filenumber%* previously opened with the OPEN statement. A value for *filenumber%* greater than 5 will result in a "Bad File Number" run-time error. The **CLOSE** statement does not cause a run-time error if the file is not open. A **CLOSE** statement with a list closes only those files listed: for example, **CLOSE 1, 3, 5** leaves files 2 and 4 open.

See Also EOF, KILL, OPEN, PRINT **[#]**

CONST

Description The **CONST** statement assigns the value *constdef* to the identifier *constantname*. This constant can be used anywhere in the script except inside quoted literal strings or comments.

Syntax **CONST** *constantname* = *constdef*

Comments

Constants defined with the CONST statement can be defined as any valid numeric or string expression (but not variables or array values), as long as their use does not cause a syntax error. For example, these statements are invalid:

```
CONST a$ = "constant " + a%
CONST b$ = a$ + " constant b"
```

However, numeric constants can be defined with numeric expressions, as follows:

```
CONST x = 1
CONST y = x + 1
```

Numeric constants can be valid numeric expressions, and can use other previously defined numeric constants in their *constdef*. String constants can only use a single string literal in their *constdef*. You cannot concatenate or use previously defined string constants to define string constants.

Constants are global in scope. They can be referenced from anywhere in a script. Constants cannot be defined within control structures. For example, **CONST** is not allowed within a SUB or FUNCTION, within a WHILE or FOR loop, nor within an IF or SELECT CASE construct.

See Also DIM, GLOBAL

CURDIR$

Description The **CURDIR$** function returns the current directory path for the drive specified or the current drive.

Syntax **A$ = CURDIR$ [***driveparm***]**

Returns **CURDIR$** always returns a fully qualified path name consisting of the drive letter, colon, and full path to the directory. Without a parameter, it returns the current working directory. With an argument, it returns the current directory on the drive indicated by the first character of the argument. For example:

```
A$ = CURDIR$("D")
A$ = CURDIR$("Delta")
A$ = CURDIR$("dos")
```

all return the current directory on the D drive.

A$ = CURDIR$

returns the current directory on the current drive because the function has no argument.

Comments

You cannot use **CURDIR$** in a UAE trap.

See Also CHDIR, CHDRIVE, MKDIR, RMDIR

DATETIME$

Description The **DATETIME$** function gives the system date and time.

Syntax **A$ = DATETIME$**

Returns The system date and time in the format MM/DD/YY HH:MM:SS.

DEALLOCATE

Description The **DEALLOCATE** statement frees the memory associated with pointer-var and assigns **NULL** to *pointer-var*. Attempting to use DEALLOCATE with a **NULL** pointer or a pointer to static program variables generates a run-time error.

Syntax **DEALLOCATE** *pointer-var*

Comments

DEALLOCATE can only be used to free memory which has been allocated with the ALLOCATE statement. Attempting to use **DEALLOCATE** with a pointer that points to static program variables generates a run-time error.

See Also ALLOCATE, REALLOCATE

DECLARE

Description The **DECLARE** statement declares a user-defined SUB or FUNCTION, or a SUB or FUNCTION residing in a dynamic-link library (DLL). Those routines declared from a DLL must be written using the Pascal calling convention. The **LIB** keyword indicates that the routine is in a DLL. The library name after the **LIB** keyword indicates the name of the library in which the routine resides. It must be a quoted literal string; string expressions or string variables are not allowed.

Syntax

DECLARE SUB *subname* **[**(*parmlist*)**]**
DECLARE FUNCTION *fnname* **[**(*parmlist*)**] AS** *typeid*
DECLARE SUB *subname* **LIB** "*libname.ext*" **[**(*parmlist*)**]**
DECLARE FUNCTION *fnname* **LIB** "*libname.ext*" **[**(*parmlist*)**] AS** *typeid*

Comments

All SUBs and FUNCTIONs must be declared prior to use in a script, including those defined in the script. Parameters are passed by reference to user-defined subroutines and functions and DLL subroutines and functions, except for **LONG**, **INTEGER**, and **POINTER** parameters, which are passed by value to routines residing in DLLs.

When passing a variable length string to a DLL routine, the string is first locked in place in the local data segment, and a far pointer to the first byte of the string is passed. The string is automatically null-terminated. If the DLL modifies the string, it should not result in a longer string than originally passed. If the string is shortened, the string variable is changed to reflect the new length and value upon return from the DLL.

If you use function names with type identifiers, you can leave off the **AS** *typeid* clause.

When declaring a DLL function, the *parmlist* for the SUB or FUNCTION has the following syntax:

(*param* **[AS [***type***| ANY]]**, ...)

The **AS ANY** clause is only legal in parameter lists for DLL functions and subroutines. When declaring a user-defined function or subroutine, you cannot declare a parameter **AS ANY.** Declaring a variable **AS ANY** turns off parameter type checking for that parameter.

See Also FUNCTION, SUB

'$DEFINE

Description The **'$DEFINE** metacommand adds a symbol to the symbol definition table.

Syntax **[REM | ']$DEFINE** *symbol*

Comments

Symbols can consist of up to 31 alphanumeric characters. Symbol names are case insensitive. The '$DEFINE metacommand can be used in association with the following conditional statements: **'$IFDEF**, **'$IFNDEF**, **'$ELSEIFDEF**, **'$ELSEIFNDEF**, **'$ELSE**, **'$ENDIF**, **'$UNDEF**.

Note that **$DEFINE** is a metacommand tha this processed during the scanning step of compilation. It is therefore not a statement.

See Also '$UNDEF

DIM

Description The **DIM** statement declares variables or arrays of variables of the associated data types. If a subscript is given, an array of variables of the given type is allocated. All arrays are zero-based, so a subscript of 10 allocates space for 11 elements, 0 to 10.

Syntax **DIM** *variable* **[**(*intconst*)**] AS** *typeid* **[**, ...**]**
DIM *variable* **[**(*intconst*)**] AS POINTER TO [POINTER TO]**... *typeid* **[**, ...**]**

Comments

The **DIM** statement gives the declared variables a scope *id* such that they are local to the current module of code. If used outside any SUB, FUNCTION, or TRAP, the variable is local to the main-line code of the script. If used inside a SUB, FUNCTION, or TRAP, the variable is local to that routine only. To declare variables that are global to the entire script, use the GLOBAL statement.

The *typeid* can represent any type, including user-defined types. The intrinsic data types in Test Driver are **INTEGER**, **LONG**, **STRING**, and STRING * n (for fixed-length strings). The **DIM** statement cannot be used to dimension a variable that has already been used in the current scope (SUB, FUNCTION, TRAP, or main-line code). A "Duplicate Definition" parse-time error occurs if the variable has already been defined.

Note that you do not need to use the **DIM** statement for simple variables of intrinsic data types (except fixed-length strings). If you use a variable without declaring it with a **DIM** or GLOBAL statement, it defaults to a **LONG** variable. If the variable has one of the Basic type identifier characters appended to it, then the variable automatically defaults to that type. Type identifier characters are % for **INTEGER**, $ for **STRING**, and & for **LONG**.

The **AS POINTER TO** form of **DIM** dimensions a pointer to a variable of type *typeid*. Pointers are strictly type checked.

See Also ALLOCATE, CONST, GLOBAL

ECHO

Description The **ECHO** statement echoes printed text to the debug terminal. **ECHO ON** causes all text printed to the viewport to be echoed to the debug terminal. **ECHO OFF** disables echoing to the debug terminal.

Syntax **ECHO [ON | OFF]**

Comments

Anything printed with **ECHO** on goes to the Test Driver viewport and to the device connected the AUX port of the computer.

See Also PRINT, VIEWPORT

END

Description If no keyword is given, the **END** statement stops execution of the Test Driver script and begins calling subroutines identified with the ON END statement. When all subroutines identified with the ON END statement have been processed, the **END** statement terminates the current block of the type specified by the given keyword. If **END** is executed in a subroutine being called as part of the ON END list, then ON END processing is terminated.

Syntax **END [IF | SELECT | SUB | FUNCTION | TRAP]**

See Also ON END, STOP

ENVIRON$

Description The **ENVIRON$** function returns the contents of the specified environment variable.

Syntax **A$ = ENVIRON$** (*environmentstring$*)

Returns An operating system environment string. The argument *environmentstring$* is a string constant that contains the name of an environment variable. The case of *environmentstring$* is important and must be uppercase in almost all circumstances. For example:

```
A$ = ENVIRON$("PATH")
```

returns the PATH environment variable, but

```
A$ = ENVIRON$("path")
```

returns an empty string.

One exception is the WINDIR environment variable under Windows. This environment variable is always lowercase.

If you specify an environment variable that cannot be found in the environment-string table, **ENVIRON$** returns an empty string. Otherwise, **ENVIRON$** returns the text assigned to the environment variable; that is, the text following the equal sign in the environment-string table for that environment variable.

EOF

Description The **EOF** function gives the end-of-file (EOF) status of a file.

Syntax **Ret% = EOF** (*filenum%*)

Returns An integer value indicating the **EOF** status of the file associated with *filenum%*. The return value is **TRUE** if the file is currently at end-of-file.

Comments

The file must have been previously opened with the OPEN statement.

See Also CLOSE, KILL, LINE INPUT, OPEN, PRINT

ERF

Description The **ERF** variable is a global string containing the filename of the script file in which the last trappable run-time error occurred.

Syntax *StringVar$* = ERF
ERF = *StringVar$*

Comments

The **ERF** variable contains the filename of the last trappable error that occurred. The combination of ERL, **ERF** and ERR can be used in logging information about run-time errors in a Test Driver script.

When a script starts, **ERF** contains a null string. Like any other variable, it can be assigned a value.

See Also ERL, ERR, ERROR, ERROR$, ON ERROR

ERL

Description The **ERL** variable is a global integer containing the line number of the line of code that caused the last trappable run-time error that occurred.

Syntax *IntVar* = **ERL**
ERL = *Integer-expression*

Comments

The **ERL** global variable will always contain the line number where the last trappable run-time error occurred. The line number is relative to the beginning of the file in which the line of code exists. This means that if a script contains **'$INCLUDE** files, and an error occurs on a line in the include file, the line number in **ERL** will be relative to the beginning of the include file, not the main script.

When a script starts, **ERL** contains 0. Like any other variable, it can be assigned a value.

See Also ERF, ERR, ERROR, ERROR$, ON ERROR

ERR

Description The **ERR** variable is a global integer containing the error code of the last trappable run-time error that occurred.

Syntax *IntVar* = **ERR**
ERR = *Integer-expression*

Comments

The **ERR** global variable will always contain the error code of the last trappable run-time error that occurred. Like any other variable, it can be assigned a value.

When a script starts, **ERR** will be 0.

See Also ERF, ERL, ERROR, ERROR$, ON ERROR

ERROR

Description The **ERROR** statement generates a runtime error.

Syntax **ERROR** (*errorcode%*)

Comments

This statement simulates the occurrence of a specific Test Basic or user-defined error.

The *errorcode%* parameter is an error code. If the value is the same as an error code already used by Test Basic, the **ERROR** statement simulates the occurrence of that error. To define your own error code, use a value that is greater than any used by the standard Test Basic error codes (start at 32,767 and work down). If the **ERROR** statment specifies an error message that is not used by Test Basic, the message User-defined error is displayed.

See Also ERF, ERL, ERR, ERROR$, ON ERROR

ERROR$

Description The **ERROR$** function returns the error message for an error code.

Syntax **A$ = ERROR$[**(*errorcode%*)**]**

Returns The error message that corresponds to a given error code.

Comments

The *errorcode%* parameter refers to a run-time error code. It must be a positive integer value. If *errorcode%* is omitted, ERROR$ returns the error message of the most recent run-time error.

See Also ERF, ERL, ERR, ERROR, ON ERROR

EXISTS

Description The **EXISTS** function checks to see if a file exists.

Syntax EXISTS (*filespec$*)

Returns An integer value indicating whether or not a file matching the specification given in *filespec$* exists. The return value is **TRUE** if such a file exists, and **FALSE** if not. The *filespec$* parameter must be a valid MS-DOS filename. You can include the MS-DOS wildcards * and ? as part of the filename. (Wildcards behave exactly like the wildcards in the MS-DOS DIR command, except that **EXISTS** does not detect subdirectories directly.)

You can use **EXISTS** to check if a subdirectory exists by checking for the always-present logical file, NUL:

```
IF EXISTS ("C:\TEST\NUL") THEN
  ...
```

If the NUL file exists in this location, then the subdirectory TEST also exists, and the **EXISTS** function returns TRUE. If the NUL file doesn't exist, then the subdirectory TEST does not exist, and the **EXISTS** function returns FALSE.

See Also KILL, OPEN, SETFILE

EXIT

Description The **EXIT** statement terminates execution of the current block identified by the given keyword and goes to the next executable statement.

Syntax **EXIT [FOR | SUB | FUNCTION | TRAP | WHILE]**

FOR... NEXT

Description The **FOR... NEXT** statement repeats a block of code a specified number of times.

Syntax **FOR** *var* = *intexp1* **TO** *intexp2* **STEP** *intexp3*
NEXT [*var***[**, *var***]**...**]**

or:

FOR *strvar$* **IN FILELIST [SORTED BY [NAME|EXTENSION]]**
NEXT [*strvar$***[**, *strvar$***]**...**]**

Comments

The **FOR** statement can keep track of the index value in an **INTEGER** or **LONG** variable to count the number of repetitions (integer syntax), or it can repeat a block of code once for each file that exists in a file list (string syntax).

In integer syntax, the index variable *var* starts at *intexp1* and is incremented by *intexp3* each time through the loop until it is equal to *intexp2*. All code between the **FOR** statement and the corresponding **NEXT** is executed until the index variable exceeds *intexp2* if *intexp3* > 0 or goes below *intexp2* if *intexp3* < 0. In string syntax, the string counter variable *strvar$* (which must be of type **STRING**; fixed-length strings are not allowed) is set to the first file in the file list, and the code up until the corresponding **NEXT** statement is executed. Then *strvar$ i*s set to the next file, and this is continued until all files in the file list have been processed. The file list can optionally be sorted prior to cycling through it. If the **SORTED** keyword is provided, the list can be sorted either by **NAME** or by **EXTENSION**, depending on the supplied keyword.

Both versions of the **FOR** statement can be terminated with the **EXIT FOR** or the **NEXT** statement. The index variable after the **NEXT** is optional---the **NEXT** is automatically matched up with the most recent **FOR** construct. The **NEXT** statement is used to terminate blocks opened by both versions of the **FOR** statement. To ensure proper stack cleanup, you should not **GOTO** out of a **FOR** statement.

When using the file list version of the **FOR** statement, the files are assigned to the string counter variable in fully qualified path name form. If you are only interested in the filename or extension, or any other section of the filename, use the SPLITPATH statement to break the filename up into its respective parts.

The **STEP** clause specifies how much *var* is incremented on each iteration. The *intexp3* parameter can also be a negative number, allowing you to decrement the loop counter, as follows:

```
FOR i = 10 TO 1 STEP -1
```

See Also END, EXIT, SETFILE, WHILE... WEND

STATIC FUNCTION

Description The **STATIC FUNCTION** statement begins the definition block of a user-defined function.

Syntax **STATIC FUNCTION fnname [**(*parmlist*)**]** AS *typeid* **[function code block]**
END FUNCTION

Comments

If you use function names with type identifiers, you can leave off the **AS** *typeid* clause. Parameters declared in the parmlist declaration section can be defined using type identifier characters or the **AS** clause. The return type must only be of intrinsic, rather than user-defined, types. The return type also cannot be a fixed-length string.

See Also DECLARE, SUB

FREEFILE

Description The **FREEFILE** function returns the next available file number.

Syntax **fnum = FREEFILE**

Returns This function returns a long value with the next unused file number, or -1 if there are no available file numbers.

Comments

Use **FREEFILE** when you need to supply a file number and you want to ensure that the file number is not already in use. **FREEFILE** returns the lowest available file number. You can have a maximum of 5 file numbers.

See Also OPEN

GLOBAL

Description The **GLOBAL** statement declares global variables or arrays. This means they can be accessed inside **subs**, **functions**, and TRAP**s** anywhere in the script.

Syntax **GLOBAL** *variable* AS *typeid* **[**, ...**]**

Comments

The **GLOBAL** statement must appear before any **subs**, **functions**, or **TRAPs** that use the declared variables. The declaration rules are the same as those for the **DIM** statement. Unlike the **DIM statement, GLOBAL** should not appear inside a **SUB**, **function**, or **TRAP (this causes a parse-time error). See the DIM** statement for more details on declaring variables.

See Also ALLOCATE, CONST, DIM

GOSUB

Description The **GOSUB** statement jumps to the code section specified by label. When the code section executes the corresponding **RETURN statement, execution resumes at the statement following the GOSUB** statement.

Syntax **GOSUB** *label*

Comments

Labels are identifiers followed by a colon, and must appear as the only item on a line. No line numbers are supported. **GOSUB** statements can be nested up to 16 levels deep.

You cannot **GOSUB** to a label outside the current code module, or the current **sub**, **function**, or **TRAP. Likewise, you cannot use GOSUB** to jump into a **sub**, **function**, or **TRAP from the mainline code.**

See Also STATIC function, GOTO, ON ERROR, STATIC SUB

GOTO

Description The **GOTO** statement unconditionally jumps to the first statement following the specified label.

Syntax **GOTO** *label*

Comments

Labels are identifiers followed by a colon, and must appear as the only item on a line. No line numbers are supported.

You can not **GOTO** to a label outside the current code module, or the current **subroutine**, **function**, or TRAP. Likewise, you cannot use **GOTO** to jump into a **subroutine**, **function**, or **TRAP** from the mainline code.

See Also **function**, GOSUB, ON ERROR, **subroutine**

IF...THEN

Description The **IF...THEN** statement executes a block of code if the given condition is TRUE, or skips to the next **ELSEIF** block (if one is present), the **ELSE** block (if one is present), or the corresponding **ENDIF** statement.

Syntax

IF *condition* **THEN**
    [*statementblock***]**
[[ELSEIF *condition* **THEN**
    [*statementblock***]]**...**]**
[ELSE
    [*statementblock***]]**
END IF

Comments

A TRUE condition is indicated by a nonzero numeric expression. Relational operations such as > and = are numeric operations that return 0 for FALSE and -1 for TRUE.

Single-line **IF...THEN** constructs are not allowed.

An **IF...THEN** construct may have any number of **ELSEIF** blocks. A single **ELSE** block is allowed, and, if present, must follow all **ELSEIF** blocks. The first block in the **IF...THEN** construct that evaluates to TRUE is executed, and then control is passed to the **ENDIF** statement.

INCLUDE metacommand

Description The **'$INCLUDE** metacommand inserts the contents of the specified files at the location of the **'$INCLUDE** metacommand.

Syntax **'$INCLUDE** *filename*

Comments

'$INCLUDE is a metacommand that is processed during the scanning step of compilation. It is therefore not a statement. The *filename* can be any valid MS-DOS pathname.

INSTR

Description The **INSTR** function looks for a specified string within another string.

Syntax **INSTR** (**[***start%*,**]** *strexp1$*, *strexp2$*)

Returns The character position of the first occurrence of *strexp2$* in *strexp1$*, optionally starting at character location *start%*.

Comments

INSTR returns zero (0) for any of the following conditions:

- *strexp2$* is not found in *strexp1$*
- *strexp2$* is longer than *strexp1$*
- *strexp1$* is an empty string.
- *start%* is greater than the length of *strexp1$*

If *strexp2$* is a null string, **INSTR** returns 1. If a negative value for *start%* is given, then **INSTR** generates a trappable run-time error.

See Also MID$, LCASE$, LEN, LTRIM$, RTRIM$, UCASE$

HEX$

Description The **HEX$** function returns a string representing the hexadecimal value of a decimal integer.

Returns A string that represents the hexadecimal value of the decimal argument.

Comments

HEX$ returns a string of up to eight hexadecimal characters. You can directly represent hexadecimal numbers by preceding numbers in the proper range with **&H**. For example, **&H10** represents decimal 16 in hexadecimal notation.

See Also STR$

KILL

Description The **KILL** statement deletes the file(s) matching the file specification, *strexp$*. Wildcard specifications are allowed in *strexp$* . (Wildcards behave exactly like the wildcards in the MS-DOS DIR command, except that **KILL** does not delete subdirectories.)

Syntax **KILL** *strexp$*

Comments

If you delete files with the **KILL** statement while performing file list operations with the SETFILE statement, use the CLEARLIST statement to immediately clear the file list and then recreate the list with the SETFILE statement.

See Also EXISTS, NAME, OPEN

LCASE$

Description The **LCASE$** function returns a copy of a string with the uppercase characters converted to lowercase.

Syntax **A$ = LCASE$** (*strexp$*)

Returns The contents of *strexp$* with all uppercase letters converted to lowercase.

See Also INSTR, LEN, LTRIM$, MID$, RTRIM$, UCASE$

LEN

Description The **LEN** function gives the length of a string.

Syntax **LEN** (*strexp$*)

Returns The length of *strexp$*.

Comments

None.

See Also INSTR, LCASE$, LTRIM$, MID$, RTRIM$, UCASE$

LINE INPUT

Description The **LINE INPUT** statement reads a line from the text file associated with *intexp%* into *strvar$*.

Syntax **LINE INPUT** *#intexp%, strvar$*

Comments

The file specified must have been previously opened with the OPEN statement for **LINE INPUT** mode. Also, *strvar$* must be of type **STRING**; fixed-length strings are not valid for use with the **LINE INPUT** statement.

See Also CLOSE, EOF, KILL, OPEN, PRINT

LTRIM$

Description The **LTRIM$** function returns a copy of the string with the leading blanks removed.

Syntax **A$ = LTRIM$** (*strexp$*)

Returns The contents of *strexp$* with all leading blanks removed.

See Also INSTR, LCASE$, LEN, MID$, RTRIM$, UCASE$

MID$

Description The **MID$** function creates a substring of specified length from a given string.

Syntax **A$ = MID$** (*strexp$*, *intexp1%* **[**, *intexp2%***]**)

Returns A substring of *strexp$* starting at character location *intexp1%* that is *intexp2%* characters long, or up to the end of *strexp$* if *intexp2%* is longer than the length of the string, or is not given.

See Also INSTR, LCASE$, LEN, LTRIM$, RTRIM$, UCASE$

MKDIR

Description The **MKDIR** statement creates a new directory.

Syntax **MKDIR** *pathname$*

Comments

The argument *pathname$* is a string expression that specifies the name of the new directory to create. The *pathname$* must have fewer than 128 characters.

The *pathname$* uses this syntax:

[*drive:***]***directory***[**...**]**

The argument *drive* is an optional drive specification; the argument *directory* is a directory name.

The **MKDIR** statement works like the MS-DOS MKDIR command. However, you cannot shorten **MKDIR** to **MD**, as you can with MS-DOS.

You can use **MKDIR** to create a directory with a name that contains an embedded space. Although you may be able to access that directory with some applications, you will be unable to remove it with standard operating system commands. You can remove such a directory using the RMDIR statement from within Test Driver.

See Also CHDIR, CHDRIVE, CURDIR$, RMDIR

NAME

Description The **NAME** statement renames files or directories.

Syntax **NAME** *oldname$* AS *newname$*

Returns Any error caused by the **NAME** statement generates a FILE I/O ERROR.

Comments

You cannot rename files or directories across drives.

See Also KILL and EXISTS

NULL

Description The **NULL** function generates a null pointer.

Syntax **NULL**

Returns Returns a null pointer that can be assigned to any pointer variable.

Comments

The **NULL** function returns a null pointer that can be assigned to any pointer data type or passed to subroutines or functions that take pointers or user-defined structures as parameters.

The **NULL** function is the only way to pass a null pointer to a variable when passing parameters to functions in DLLs such as the Windows API. The following table shows the results of passing **NULL** to a function or subroutine in the MSTEST DLLs.

Type	**Result Passed to FUNCTION or SUB**
String ($)	**NULL** pointer
Integer (%)	Integer Value (0)
Long (&)	Long value (0)
Type Struct	**NULL** pointer
POINTER	NULL pointer
ANY	**NULL** pointer

Passing **NULL** to a user-defined subroutine or function results in a type mismatch for all types except **POINTER**, which passes a **NULL** pointer.

See Also VARPTR

ON END

Description The **ON END** statement adds one or more subroutines to a list of subroutines to be called when the **END** statement is executed.

Syntax **ON END** *subname* **[[,** *subname***]...]**

Comments

The **ON END** statement defines a list of subroutines to be called when a script reaches the last executable statement or when a script encounters an END statement. The **SUB** name must be a user-defined SUB subprogram. It cannot be a user-defined **function** or a **function** or SUB in a DLL. A subroutine used with **ON END** must be declared before the **ON END** statement and cannot have any parameters. This means that the following are not legal:

```
DECLARE SUB sub1(x%)                  ' Cannot use a SUB with parameters.
DECLARE function func1() AS INTEGER   ' Cannot use a function.
DECLARE SUB DLLSub LIB "MYLIB.DLL" () ' Cannot use a routine in a DLL.
ON END SUB1, funct1, DLLSub, SUB2
DECLARE SUB sub2  ' Routines must be declared before the ON END  statement.
' The rest of the script goes here.
```

The following is legal:

```
DECLARE SUB sub1
DECLARE SUB sub2
ON END SUB1, SUB2
' The rest of the script goes here.
```

If more than one **ON END** statement is used in a script, the SUBs are added to the list of SUBs to be called in the reverse order that the ON END statements occur, in "last in, first out" order.

The following causes a call to **SUB2**, then **SUB3**, then **SUB1** when the script ends. If a SUB is used with **ON END** more than once, then it is called more than once.

```
ON END SUB1
ON END SUB3
ON END SUB2
```

The following series of ON END statements would cause a call to SUB1, then SUB3 then SUB1 again as the script ends

```
ON END SUB1
ON END SUB3
ON END SUB1
```

There are two exceptions in which some or all of the SUBs in the **ON END** list may be called. A script ends after the the last executable line of code is executed or when the END statement is encountered. However, the STOP statement causes a script to end without calling any of the SUB**s** in the **ON END** list.

All SUB**s** in the **ON END** list are called in the reverse order that they are added to the list when the program ends. However, if one of the SUBs in the list contains either an END statement or a STOP statement, the rest of the SUB**s** in the list will not be called. The script will stop executing at that point in that SUB.

You can place up to eight SUB**s** on the **ON END** list. Once added, they cannot be removed programmatically.

Example

```
'=========================================
declare sub sub1
declare sub sub2
declare sub sub3
declare sub sub4

on end sub4, sub3, sub1, sub2, sub1

viewport clear

'stop
' Un-comment this to prevent any ON END subroutines
' from executing.

end

sub sub1
  print "in sub1"
end sub

sub sub2
  print "in sub2"
end sub

sub sub3
  print "in sub3"
  STOP
  END
end sub

sub sub4
  print "in sub4 -- This should never print."
end sub

'=========================================
```

This example would cause the following to be printed to the viewport.

```
in sub1
in sub2
in sub1
in sub3
```

See Also END, STOP

ON ERROR

Description The **ON ERROR** statement lets a script trap and recover from run-time errors. If no **ON ERROR** statement is used, any run-time error that occurs is fatal; that is, Test Driver generates an error message, and stops execution of the script.

Syntax **ON ERROR GOTO [** *linelabel* **|** 0 **]**

Comments

Error trapping is global only. This means you cannot use the **ON ERROR GOTO** statement within a procedure. In addition, the *linelabel* where the error trap starts may not be inside of a procedure. However, when errors occur within a procedure, they will be trapped by the global error handler. When the error handler executes a RESUME statement, execution resumes within the procedure where the error occurred.

ON ERROR GOTO 0 is a special case of the **ON ERROR** statement which disables error trapping in the script. After executing this statement, any run-time errors that occur will stop execution of the script unless another **ON ERROR GOTO** statement enables error trapping again.

See Also ERF, ERL, ERR, ERROR, ERROR$, RESUME

OPEN

Description The **OPEN** statement opens the file specified by *strexp$* in the given mode, and associates the file with *intexp%*.

Syntax **OPEN** *strexp$* **FOR [INPUT|OUTPUT|APPEND] AS [#]** *intexp*%

Comments

If opened for **INPUT** mode, the file must exist, or a "File Not Found" run-time error occurs. If opened for **OUTPUT** mode, if the file exists, it is overwritten. If opened for **APPEND** mode, the file is created if it does not already exist. If it does exist, any PRINT **#** statements used on the file are appended to the end of the file. The *intexp%* parameter must be between 1 and 5.

See Also CLOSE, EOF, EXISTS, KILL, PRINT **#**

PAUSE

Description The **PAUSE** statement displays *strexp$* in a message box and waits for the user to acknowledge.

Syntax **PAUSE strexp$**

Comments

The *strexp$* parameter appears in a message box with an OK button.

See Also END,STOP and SLEEP

PRINT

Description The **PRINT** statement displays information to the viewport, or sends information to the file associated with *intexp*.

Syntax **PRINT [[#***intexp*,**]** *exp* **[[**;|,**]** *exp*...**] [**;|,**]]**

Comments

The **PRINT** statement prints each expression in the expression list, and the semicolon or comma (if given) determines what extra characters should be printed after each expression. The semicolon causes no extra characters to be printed; the comma causes a tab character to be printed; and if neither a semicolon nor a comma is given, a carriage-return/line-feed pair is printed. Note that if more than one expression is printed and they are not separated by either a semicolon or a comma, the **PRINT** statement assumes a semicolon is at the end of the statement. **PRINT** with no expressions simply prints a carriage-return/line-feed pair.

If printing to a file, intexp% must represent a file that has been previously opened with the OPEN statement for OUTPUT or APPEND mode. The file number must be between 1 and 5, or a "Bad File Number" run-time error occurs.

See Also ECHO, OPEN, VIEWPORT, CLOSE

RANDOMIZE

Description The **RANDOMIZE** statement seeds the random number generator with *intexp%*. The RND function uses this value as its starting location for random number generation.

Syntax **RANDOMIZE** *intexp*%

See Also RND

REALLOCATE

Description The **REALLOCATE** statement resizes the memory buffer associated with pointer-var. **REALLOCATE** attempts to shrink or grow the memory buffer to *num-items* * *size* bytes, where *size* is the size in bytes of the data type *pointer-var* points to.

Syntax **REALLOCATE** *pointer-var*, *num-items*

Comments

REALLOCATE can only be used to resize memory which has been allocated with the **ALLOCATE** statement. Attempting to use **REALLOCATE** with a **NULL** pointer or a pointer that points to static program variables generates a run-time error.

See Also ALLOCATE, DEALLOCATE

REM

Description The **REM** statement lets you include a comment in a script. You can use a single quotation mark in place of the keyword **REM**. **REM** can also introduce a metacommand (a special instruction to the compiler) such as **$DEFINE**.

Syntax **[REM | ']** *comment*

Comments

comment is text that has any combination of characters.

RESUME

Description The **RESUME** statement resumes program execution when an error-trap routine is finished handling the error.

Syntax **RESUME [NEXT |** *line-label* **]**

Comments

RESUME NEXT causes execution to resume with the statement immediately following the one that caused the error. **RESUME** line-label causes execution to resume at a line label. The argument *line-label* must be in the main code of the script. It cannot be in a subroutine or function.

See Also ON ERROR

RETURN

Description The **RETURN** statement returns to the statement following the most recent GOSUB statement.

Syntax **RETURN**

Comments

If a **RETURN** statement is executed before a corresponding GOSUB statement, a "RETURN without GOSUB" run-time error occurs. GOSUB/**RETURN** pairs may be nested up to 16 levels. If more levels are attempted, a "GOSUB Stack Overflow" run-time error occurs.

Unlike some Basic interpreters, you cannot return to a specific label. **RETURN** passes control back to the statement following the most recently executed GOSUB statement only.

See Also GOSUB

RMDIR

Description The **RMDIR** statement removes an existing directory.

Syntax **RMDIR** *pathname$*

Comments

The argument *pathname$* is a string expression that specifies the name of the directory to remove and must have fewer than 128 characters.

pathname$ uses this syntax:

[*drive:***]***directory***[**...**]**

The argument *drive* is an optional drive specification; the argument *directory* is a directory name.

The **RMDIR** statement works like the MS-DOS **RMDIR** command. However, you cannot shorten **RMDIR** to **RD** as you can with MS-DOS.

See Also CHDIR, CHDRIVE, CURDIR$, MKDIR

RND

Description The **RND** function generates a pseudo-random number between 0 and 32767.

Syntax ***i%*** = **RND**

Returns A pseudo-random number between 0 and 32,767.

See Also RANDOMIZE

RTRIM$

Description The **RTRIM$** function returns a copy of the string with the trailing blanks removed.

Syntax **A$ = RTRIM$** (*strexp$*)

Returns The contents of *strexp$* with all trailing blanks removed.

See Also INSTR, LCASE$, LEN, LTRIM$, MID$, UCASE$

RUN

Description The **RUN** function runs a program.

Syntax **ret% = RUN** (*strexp$*)

Returns After spawning the process identified in *strexp$*, the return value indicates the success or failure of the operation.

Comments

The function returns a value greater than 32 if the process was successfully spawned. Otherwise, the process was not started and the return value indicates the error that occurred. See the Windows SDK reference on **WinExec** for a description of the error return values. The script continues as soon as the child process is spawned, and the two processes run asynchronously. *Strexp$* cannot exceed 128 characters.

The **RUN** function is essentially the same as the **RUN/NOWAIT** statement.

See Also RUN STATEMENT, SHELL

RUN

Description The **RUN** statement executes the program specified by *strexp$*. The optional **NOWAIT** parameter allows the process to run asynchronously (if supported by the host operating system). If not given, the child process is allowed to complete before the script continues on to the next statement.

Syntax **RUN** *strexp$* **[**, **NOWAIT]**

Comments

Strexp$ cannot exceed 128 characters.

See Also RUN FUNCTION, SHELL

SELECT CASE

Description The **SELECT CASE** statement executes one of several statement blocks depending on the value of the given expression and the expressions given on the **CASE** statements.

Syntax

SELECT CASE *expression*
[CASE *exp* **[TO** *intexp***] [**, *exp* **[TO** *intexp***]**...**]**
    [*statementblock***]**...**]**
[CASE IS *relational-operator expression***]**
[CASE ELSE
    [*statementblock***]]**
END SELECT

Comments

The expression on the **SELECT CASE** statement can be any expression of either **INTEGER**, **LONG**, or **STRING** type, so long as the expressions in the following **CASE** statements are of the same type.

The **TO** keyword on the **CASE** statement is only valid when used with numeric **SELECT CASE** expressions. The expression on the left side of the **TO** keyword must be the smaller of the two expressions, or that **CASE** clause never evaluates to **TRUE**.

If more than one expression clause is given on the **CASE** statement, their values are combined with a logical **OR** together to determine the truth value of the entire **CASE** block. The first **CASE** block that is executed transfers control to the **END SELECT** statement when it is finished. The **CASE ELSE** block is executed only if none of the expression clauses on the **CASE** blocks above it were true. The **CASE** ELSE block must be the last block in the **SELECT CASE** construct.

See Also IF...THEN

SETFILE

Description The **SETFILE** statement adds (**ON**) or subtracts (**OFF**) all files matching the specification given in *strexp$* to or from the file list.

Syntax **SETFILE** *strexp$*, **[ON | OFF]**

Comments

Any files matching the specification that are already present in the file list are not duplicated. Wildcard specifications are allowed in *strexp$*, both for **ON** and **OFF** operations. (Wildcards behave exactly like the wildcards in the MS-DOS DIR command, except that **SETFILE** does not list subdirectories.)

If you delete files with the KILL statement while performing file list operations with the **SETFILE** statement, use the CLEARLIST statement to immediately clear the file list and then re-create the list with the SETFILE statement.

See Also CLEARLIST, FOR, KILL

SHELL

Description The **SHELL** statement passes *strexp$* to the MS-DOS command processor for execution.

Syntax **SHELL** *strexp$*

Comments

The **SHELL** statement opens a MS-DOS box to perform the operation given in *strexp$*. The script does not continue execution until this MS-DOS box has completed its task and terminates. *Strexp$* cannot exceed 110 characters.

See Also RUN

SLEEP

Description The **SLEEP** statement suspends execution of the script until after *intexp%* number of seconds, or until you issue a BREAK event by pressing the ESC key.

Syntax **SLEEP** (*intexp%*)

Comments

The **SLEEP** statement can be used to stop execution of script code, but not leave **RUN** mode. This allows all traps to remain active while the script has nothing to do. For example, you could run a script that traps UAE conditions and then put it to sleep. The script will wait in the background until a UAE occurs, at which time it wakes up and continues with whatever instructions are in the script.

If an integer expression is given, the **SLEEP** statement suspends the script for the given number of seconds. If *intexp%* is 0 or not provided, the script is suspended indefinitely. You can stop a suspended script by selecting Break from the Run menu.

See Also END, PAUSE, STOP

SPLITPATH

Description The **SPLITPATH** statement splits the path name given in *strexp$* into its respective parts. The drive specification goes in *drv$*, the directory into *dir$*, the base filename into *filename$*, and the extension into *ext$*.

Syntax **SPLITPATH** *strexp$*, *drv$*, *dir$*, *filename$*, *ext$*

Comments

The four target strings must be of type **STRING**. Fixed-length string arguments are not valid targets for the **SPLITPATH** statement.

See Also SETFILE

STOP

Description The STOP statement terminates a TestBasic script.

Syntax **STOP**

Comments

STOP terminates program differently than the **END** statement. **END** allows **ON END** processing of subroutines, whereas **STOP** does not. Both **STOP** and **END** close any open files before terminating the script.

See Also END, PAUSE, RESUME

STRING$

Description The **STRING$** function returns a string whose characters all have the given ASCII charcode.

Syntax **A$ = STRING$** (*number%*, *charcode%*)
A$ = STRING$ (*number%*, *StringExpression$*)

Returns A string whose characters all have the given ASCII charcode% or a string whose characters are all the same as the first character of *StringExpression$*.

Comments

The *number%* parameter is the length of the string to return. If number% is 0, a null string is returned. If there is not enough string space to hold the string, then an "Out of string space" run-time error is generated.

The *charcode%* parameter is the ASCII code of the character used to build the string. It is a numeric expression which must be between 0 and 255. If *charcode%* is greater than 255 then the character used to build the string will be ASCII character (*charcode%* MOD 256).

The *stringExpression$* parameter is the string expression whose first character is used to build the returned string. If a null string is given, an "Illegal function call" run-time error is generated.

See Also CHR$, HEX$, STR$, VAL

STR$

Description The **STR$** function returns an integer expression as an ASCII string.

Syntax **A$ = STR$** (*intexp%*)

Returns A string representing *intexp%* in ASCII form. If the number is positive, the string is padded with a single space at the beginning. If negative, the string is returned with the minus (-) sign at the beginning and no leading space.

See Also HEX$, VAL, CHR$, STRING$

STATIC SUB

Description The **STATIC SUB** statement begins the definition of a user-defined SUB.

Syntax **SUB** *subname* **[**(*parmlist*)**]**
[*subprogram block***]**
END SUB

Comments

Parameters declared in the parmlist declaration section can be defined using type identifier characters or the AS clause. Parameters must only be of intrinsic types, excluding fixed-length strings.

See Also FUNCTION

TIMER

Description The **TIMER** function gives the number of seconds (in hundredths) since midnight.

Syntax **secnds% = TIMER**

Returns The number of hundredths of a second elapsed since midnight, in **LONG** format. You should always assign the return value to a **LONG** variable.

See Also DATETIME$

TRAP

Description The **TRAP** statement allows definition of a block of code to be executed when the event defined by *trapname* (which appears in the DLL LIBNAME.EXT) occurs.

Syntax **TRAP** *trapname* FROM "*libname.exe*"
[*trap code block***]**
END TRAP

Comments

Care should be taken when writing **TRAP** service routines. Depending upon the event that you are trapping, certain actions that may be "acceptable" under normal circumstances may be hazardous inside **TRAP** routines.

UCASE$

Description The **UCASE$** function returns a copy of a string with the lowercase characters converted to uppercase.

Syntax **A$ = UCASE$** (*strexp$*)

Returns The contents of *strexp$* with all lowercase letters converted to uppercase.

See Also INSTR, LCASE$, LEN, LTRIM$, MID$, RTRIM$

VARPTR

Description The **VARPTR** function generates a far pointer to a variable.

Syntax **VARPTR** (*variable*)

Returns A far pointer to the given variable.

Comments

The variable can be almost any Test Driver data type: **LONG**, **INTEGER**, **STRING ***, **POINTER**, or user-defined **TYPE**. However, the variable cannot be a **STRING** since this would return the address of the string descriptor not of the string contents. Using **VARPTR** with a variable-length string produces a type mismatch error.

'$UNDEF

Description The **'$UNDEF** metacommand removes symbols from the symbol definition table that had been previously defined with the **'$DEFINE** metacommand.

Syntax **'$UNDEF** *symbol*

Comments

Symbols can consist of up to 31 alphanumeric characters. Symbol names are case insensitive. The "$UNDEF metacommand can be used in association with the following conditional statements: **'$IFDEF**, **'$IFNDEF**, **'$ELSEIFDEF**, **'$ELSEIFNDEF**, **'$ELSE**, **'$ENDIF**, **'$UNDEF**.

Note that **'$UNDEF** is a metacommand that is processed during the scannign step of compilation. It is therefore not a statement.

See Also '$DEFINE

VAL

Description The **VAL** function generates the integer value of a specified string.

Syntax **intval% = VAL** (*strexp$*)

Returns The integer value of *strexp$* or 0 if the string does not represent an integer.

See Also HEX$, STR$

VIEWPORT

Description The **VIEWPORT** statement displays, hides, or clears the viewport window.

Syntax **VIEWPORT [ON | OFF | CLEAR]**

Comments

The **VIEWPORT ON** statement displays the viewport in its most recent state, but does not give it focus.

See Also PRINT

WHILE...WEND

Description The **WHILE...WEND** statement executes a series of statements in a loop, as long as a given condition is TRUE.

Syntax **WHILE** *condition*
[*statementblock***]**...**]**
WEND

Comments

The argument *condition* is a numeric expression that evaluates as TRUE or FALSE.

If *condition* is TRUE, any intervening statements are executed until the **WEND** statement is encountered. Test Driver then returns to the **WHILE** statement and checks *condition*. If it is still **TRUE**, the process is repeated. If it is not **TRUE**, execution resumes with the statement following the **WEND** statement.

The **WHILE** statement only checks for a nonzero value. For example, if condition is 2, the **WHILE** statement will still evaluate this as TRUE.

You can nest **WHILE... WEND** loops to any level. Each **WEND** statement matches the most recent **WHILE**. When Test Driver encounters an unmatched **WHILE** statement, it generates the error message, **WHILE** without **WEND**. If Test Driver encounters an unmatched **WEND** statement, it generates the error message "WEND without WHILE".

Note Do not branch into the body of a **WHILE... WEND** loop without executing a **WHILE** statement. Doing so may cause errors or program problems that are difficult to locate.